Department of Applied Informatics
Faculty of Mathematics, Physics and Informatics
Comenius University

Pavol Vančo

Processing of Tree-Structured Data with Recursive Self-Organizing Maps

Dissertation Thesis

Supervisor: doc. Ing. Igor Farkaš, PhD.

9.2.1 INFORMATICS

Bratislava, 2010

Dedicated to my family and all my friends for believing in me.

Hereby I declare that I wrote this thesis myself with the help of no more than the referenced sources.

Vyhlasujem, že predkladaná práca je mojím originálnym dielom, ktorý som vypracoval samostatne, s použitím zdrojov uvedených v zozname literatúry.

Signed:

Abstract

VANČO, PAVOL: *Processing of Tree-Structured Data with Recursive Self-Organizing Maps.* [Dissertation thesis]. Comenius University in Bratislava. Faculty of Mathematics, Physics and Informatics; Department of Applied Informatics. Thesis advisor: doc. Ing. Igor Farkaš, PhD. Thesis defense committee: 9.2.1. Informatics. Committee chairman: prof. RNDr. Branislav Rovan, PhD. Qualification degree: Philosophiæ doctor in Informatics. Bratislava, 2010. 107 p.

The thesis deals with selected models of recursive self-organizing maps (SOMSD, MSOM and RecSOM) that have recently been extended for processing complex data types, namely tree structures. Regarding MSOM we argue that despite the commutativity operation in context computation it can distinguish between the branches of trees with permuted children, and support our theoretical claim with computer simulation. We experimentally compare the recursive SOMs using three data sets of increasing complexity. For comparison we introduce and apply six quantitative measures focusing on different aspects of the trained maps. Next, the practical usage of these recursive maps is experimentally shown on data sets encoded in XML format. Visualization and clusterization using these models is used for data mining. Batch learning for SOMs is presented with its advantages and disadvantages. It is shown experimentally when batch learning, in its current form, cannot be used, and a modified learning algorithm is proposed. Batch learning for recursive and recurrent SOMs is introduced that enables distributed computation. Finally, methods for data extraction constructed from the trained maps are introduced. When the complete data cannot be completely retrieved, the data reconstruction methods are proposed, either in a form of a lookup table or based on a feedforward neural network.

Abstrakt

VANČO, PAVOL: *Spracovanie stromových štruktúr pomocou rekurzívnych samoorganizujúcich sa máp.* [Dizertačná práca]. Univerzita Komenského v Bratislave. Fakulta matematiky, fyziky a informatiky; Katedra aplikovanej informatiky. Školiteľ: doc. Ing. Igor Farkaš, PhD. Komisia pre obhajoby: 9.2.1 Informatika. Predseda: prof. RNDr. Branislav Rovan, PhD. Stupeň odbornej kvalifikácie: Philosophiæ doctor v odbore Informatika. Bratislava, 2010. 107 s.

Práca sa zaoberá vybranými modelmi rekurzívnych samoorganizujúcich sa máp (SOMSD, MSOM a RecSOM), ktoré boli v ostatných rokoch navrhnuté na spracovanie zložitých dátových typov, konkrétne stromových štruktúr. V prípade modelu MSOM argumentujeme, že aj napriek komutatívnosti operácie vo výpočte kontextovej reprezentácie, MSOM dokáže rozlíšiť medzi vetvami stromov s permutovanými synmi, čo potvrdzujeme aj výpočtovou simuláciou. Vybrané modely rekurzívnych SOM experimentálne porovnávame na troch dátových množinách rôznej zložitosti. Za účelom porovnania definujeme a aplikujeme šesť kvantitatívnych mier, ktoré sa zameriavajú na rôzne vlastnosti natrénovaných máp. Praktické použitie týchto rekurzívnych máp ilustrujeme na dátových množinách zakódovaných v XML formáte. Vizualizácia a klasterizácia pomocou týchto modelov slúži na objavovanie znalostí v dátach. V prípade klasických SOM prezentujeme dávkové učenie s jeho výhodami a nevýhodami, poukazujeme na obmedzenia jeho použitia a navrhujeme úpravu algoritmu. V prípade rekurzívnych a rekurentných SOM tiež navrhujeme algoritmus dávkového učenia, ktoré umožňuje distribuované spracovanie dát. V závere navrhujeme metódy objavovania znalostí z natrénovanej mapy. Ak nie je možné získať úplné dáta, uvedené sú metódy rekonštrukcie dát, a to buď pomocou tabuľky alebo pomocou doprednej neurónovej siete.

Foreword

In this thesis we deal with existing recursive models of self-organizing maps, designed for processing of structured data, namely trees. We also extend one recurrent model for more complex structures. These recursive models, introduced quite recently in the literature, can provide a connectionist alternative (to symbolic models) for processing structured data, and their computational properties have yet to be appreciated.

We compare these models on three different data sets to point to the strengths and weaknesses of the presented models. We show how these models can be used in real-life application using XML format. We focus on batch learning of the classic self-organizing map and show, using an example, when batch learning should not be used. We propose two new types of batch learning for recursive models. We also propose how to use recursive models as memory by data extraction. We demonstrate how to reconstruct data from the recursive models when only incomplete information is available.

I started to work with self-organizing maps during my studies at the university. As for my diploma thesis I chose to work with recurrent models, simple recurrent networks. I decided to work on recurrent self-organizing maps with my supervisor. After the introduction into recurrent networks I moved onto more complex recursive self-organizing maps.

I would like to thank Igor Farkaš, my supervisor, for all his help and for patiently answering all my questions. I would also like to thank people who helped me with teaching as it was a great challenge. I am grateful for my family and for my closest friends who provided me with their full support every time I needed it.

Contents

List of Tables

List of Figures

List of Algorithms

Definitions

Symbol conventions

- N – map size (also the number of neurons in the map)
- X – input data set
- m – input data set size ($m = |X|$)
- $\mathbf{x}$ – input vector ($\mathbf{x} \in X$)
- n – input vector length ($n = \|x\|$)
- $\mathbf{w}_i$ – input weight vector of the i-th neuron ($i \in N$)
- $\mathbf{c}_i$ – context weight vector of the i-th neuron ($i \in N$)
- $\|\mathbf{x} - \mathbf{w}_i\|^2$ – squared Euclidean distance between two vectors
- $d(\mathbf{x}, \mathbf{w}_i)$ – distance between weight vector of the i-th neuron ($i \in N$) and the input vector, usually the squared Euclidean distance
- i^* – winner (best matching unit) for the current input, $i^* \in N$
- $dist(i, j)$ – Euclidean distance between positions of neurons i and j in the grid of the SOM model
- $\mathbf{y}$ – map output activation vector
- $h(i^*, j)$ – neighborhood function, provides proximity to the winning neuron, usually computed as the Gaussian function
- k – arity of tree
- M – structured input size, in case of unstructured data $M = 1$, in case of sequences it is the length of the sequence, in case of complete binary trees of maximal depth d $M = 2^d - 1$

- $M_{\max}$ – maximum size of the structure in the data set, for the sequences it is the length of the longest sequence, $M_{\max} = \max(M_{\mathbf{x}}|\mathbf{x} \in X)$
- τ – time constant for Temporal Kohonen map model, $0 < \tau < 1$
- $V_i(t)$ – activation of the unit i at time step t for the Temporal Kohonen map model

Naming conventions

- Epoch – the presentation of the whole data set
- SOM – Self-Organizing Map
- RecSOM – Recursive SOM
- MSOM – Merge SOM
- SOMSD – SOM for Structured Data
- TKM – Temporal Kohonen Map
- RSOM – Recurrent SOM

Chapter 1

Introduction

The world around us abounds with structures. The examples range from time series, DNA sequences, chemical structures, logical formulas, graphical objects, database entries to complex web pages containing text, pictures and links. The ability to process structured data should hence be a required feature of any information system.

In the thesis, we focus on artificial neural networks, specifically self-organizing maps that were originally designed for processing vectorial data and only recently were extended to processing sequences and tree structures. The need to compare and analyze these recurrent and recursive models arises from basic research itself (better understanding of their functionality) as well as their practical usage as data processors.

1.1 Structures

Data can be organized in different types of structures. The most general structure type is a graph.

Definition 1.1.1 *A graph is a pair $G = (V, E)$ of sets such that $E \subseteq V \times V$. The elements of V are vertices (or nodes) of the graph G. The elements of E are edges of the graph G.*

In simple words graph is a structure that has no restrictions on edges, i.e. any vertex can be connected to any vertex through an edge. Restrictions on the edges create different types of structures: trees, forests, palms, sequences, complete graphs, edgeless graphs, bipolar graphs, Hamiltonian graphs, etc. By labeling four types of graphs can be created: unlabeled, vertex-labeled, edge-labeled and labeled[1].

[1] labeled graph has both vertices and edges labeled

Words in sentence or time series are examples of sequences, the simplest data structure, consisting of connected vertices. In our case every vertex has a predecessor and a successor, i.e. it has one edge coming in the vertex and one coming out[2].

Some examples:

Sequence of numbers:	1 1 2 3 5 8 13 21 34 55 . . .
Sequence of letters:	a a b b b a a a a c c c c c c . . .
Sequence of words:	how much wood would a woodchuck chuck . . .

Definition 1.1.2 *A sequence is a connected directed acyclic graph where at most one edge is ending in any vertex and at most one edge is starting in any vertex.*

Sequences can be finite or infinite. We will be working with finite sequences as these can be processed in a finite time.

The more complicated structure we are working with is a tree.

Definition 1.1.3 *A tree is a connected acyclic graph. Vertices of degree 1 are called leaves, other vertices are called inner vertices. A root is a special vertex of a tree. If a tree has a root vertex defined it is called a rooted tree.*

In other words, a rooted tree has a root, inner vertices and leaves.

Among real-life applications processing structured data are: logo recognition, e.g. trying to differentiate company logos and other types of pictures based on structure and its properties (color, shape, etc.), language sentence processing (also with grammar structure), chemical structure visualization and categorization, etc. More information about sequences and trees can be found in Diestel (2005).

1.2 Processing of structured data

In order to process structured data in artificial intelligence we can use different approaches:

- Symbolic approach – manipulates the structured data as vertices in defined structure. Structured data is saved in the knowledge base and is processed with rules. The rules are created externally with prior knowledge. Data can be processed without any preprocessing.

[2] with the exception of the first vertex (no predecessor) and the last vertex (no successor)

- Subsymbolic approach (connectionist) – is inspired by biological principles using parallel distributed processing. Saving and processing of data is distributed among units (artificial neurons) and its distribution has to be analyzed. Learning (changing internal parameters) is done with or without external supervision.

We will concentrate on the latter approach. Among the subsymbolic approaches – neural networks, fuzzy systems, evolutionary computation, etc. – we will focus on the neural networks.

Neural networks constitute a particularly successful approach that allows learning an unknown regularity from a given set of training examples. In many domains (such as time series prediction, bioinformatics or image processing) the data is non-standard, presented in the form of sequences, trees or graphs. Although the data items can be numeric or symbolic depending on the data set. Neural networks have been applied to both types of data (Hammer, 2003). Processing the non-standard data with neural networks has followed two major directions (Hammer and Jain, 2004). One way is to use models exploiting a similarity measure adapted to non-standard data. Alternatively, non-standard data can be processed recursively with a suitable architecture of a neural network, trained in supervised or unsupervised manner.

Neural networks can be trained with providing required output (learning with teacher, supervised learning), with providing good or bad response (reinforcement learning), not providing any information about required output (learning without teacher, unsupervised learning).

Self-organization is a term used for some models using learning without teacher. Input processing is based on the characteristics of the data set. There is no need for neither external influence nor external help.

Unsupervised learning is a more difficult alternative applicable in cases where no explicit teaching signal is available. Hence, the network must learn to extract useful information from the data without feedback, being driven only by statistical properties of data. Unsupervised networks have been typically used in data mining and visualization. The most frequently used model in this category is the self-organizing map (SOM; Kohonen (1990)) that has been applied in numerous tasks ranging from web-mining to robotics (Kaski et al., 1998).

1.3 Artificial neural networks for vectorial data

In 1943 McCulloch and Pitts presented a computer simulated neuron. It was a simplified model based on brain cells. Minsky and Papert (1969) mathematically proved that one layer perceptrons can solve only linearly separable problems. Rumelhart et al. (1986) created the first continuous neuron with a learning called error back propagation. After that many types of neural networks were created and tested for various purposes, e.g. recognition, classification, approximation, extraction of attributes and data mining. Training of neural networks has usually high time complexity and therefore only relatively small numbers of artificial neurons are used in simulations.

Neuron is an important biological cell in our brain. The complexity of the brain results from the interconnection of vast numbers of the neuron cells and the rate at which they are firing (Čihák, 2004). The neuron consists of three parts: dendrites and basal dendrites, nucleus (soma), axon.

Dendrites are inputs to the neuron and axon is output from the neuron. Dendrites are mostly short and they are responsible for reception of the signal through synapses and transportation of this signal to the soma of the cell. The nucleus processes all signals from all dendrites and also from the surface of the cell and then it sends a signal through the axon if signal is strong enough. Axon is a part of the neuron with many terminals at the end. These terminals are connected to dendrites of other neurons via synapses (Fig. 1.1).

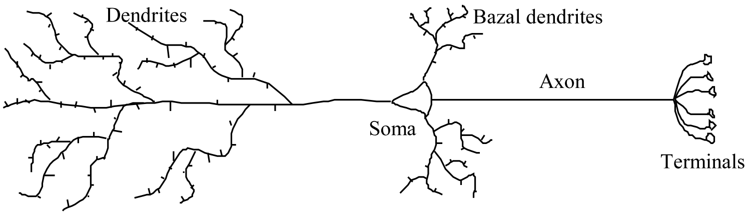

Figure 1.1: Model of neuron. Dendrites, basal dendrites are inputs, axon is output of the nucleus. Axon ends with terminals (figure taken from (Návrat et al., 2002)).

The behavior of the neuron cell begins and ends on a connection between two neurons called synapse. The end of axon from the first neuron is ended with the presynaptic terminal that is separated from the postsynaptic membrane by a synaptic cleft. The postsynaptic membrane is located mostly on the dendrites but it can be also located on the body cell. The electric signal is converted into a chemical signal in the presynaptic terminal (uses anions and cations) which is transmitted through the synaptic cleft to the postsynaptic terminal (Beňušková, 2000). There it is converted again into an electric signal. Depending on type of synapse it can inhibit or excite the signal that means increase or decrease electric potential. This signal is then transported to the soma. When the summed signals from every synapse are greater than neuron threshold, neuron fires. That means it generates a signal which is sent through axon to all other neurons with which it is connected.

The artificial neuron uses analogy from biological neural cell to simulate its work. It is a simplified model. Synapses are represented by multiplication of incoming signal with weight of the synapse, the soma is represented by a mathematical function which input is sum of all incoming signals. If this value is greater or equal than defined threshold neuron creates output:

$$y = f(\mathbf{w}^T\mathbf{x} - \theta), \tag{1.1}$$

where $\mathbf{x}$ is the input signal to the neuron, $\mathbf{w}$ is the weight vector of the neuron and parameter θ is the threshold of the neuron. This function is called bipolar binary activation function.

There are other possibilities to define the output of the neuron, one of which is the Euclidean distance in RBF model (Eq. 1.2).

$$y = \exp(-\left\|\mathbf{x} - \mathbf{w}\right\|^2 / \sigma^2). \tag{1.2}$$

Usually axons and dendrites (inputs and outputs) in artificial neuron are ideal transporters of signal. There are many types of neurons depending on complexity (simplicity) and used function: discrete neurons, sigmoid neurons, spiking neurons, BCM neurons (Návrat et al., 2002; Jedlička, 2002).

The simplest artificial neurons are discrete neurons which use bipolar binary activation function (Rosenblatt, 1958).

Sigmoid neurons use continuous activation function – sigmoid function (Eq. 1.3).

$$f(net) = \frac{1}{1 + e^{-\mathbf{net}}} \tag{1.3}$$

More models were created depending on problem which they were proposed to solve: recurrent networks for time-series prediction, principal component analysis networks for linear auto association, Hopfield network for memory representation (Kvasnička et al., 1997; Šíma and Neruda, 1997).

1.4 Self-organizing maps

Processing and visualization of multidimensional data using subsymbolic paradigm is a known task that can be solved using various techniques. In situation where input data have no required output learning without teacher can be chosen. Among clustering models Self-organizing map model has an important place.

Self-organizing map (SOM) was introduced by Kohonen (1982). The idea of the SOM originates from the topographic maps detected in human brain. These maps are projection maps of a surface of a human body just as a human body appears. Close points found on human body are represented by close neuron clusters in the brain. These maps are not uniform, i.e. the more sensible surface of body is (more neurons are present on that particular place) the larger the neuron cluster for this region is.

In its standard form, SOM has been formulated for vectorial data, i.e. for inputs belonging to a vector space of a finite and fixed dimension. SOM typically provides a topographic (nonlinear) mapping from high dimensional input space to a discrete grid of units, exploiting principles of competition and cooperation among the units.

SOM architecture consists of two parts: input and neural map. Input size is based on encoding of input. The neural map is usually one or two dimensional. Neurons in the map are placed on defined positions in a lattice. These positions are then used as excitatory while the strength of the excitation is depending on the position of the best matching unit. All inputs are interconnected with each neuron in the map (Fig. 1.2).

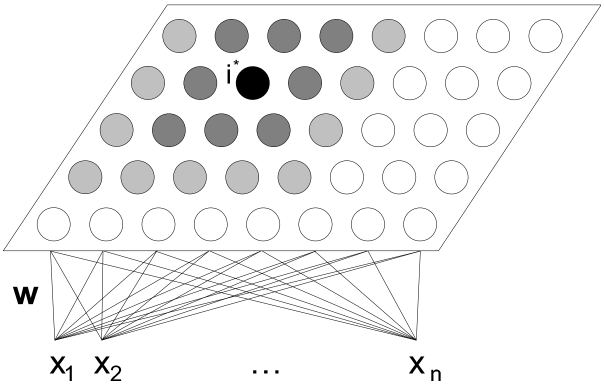

Figure 1.2: Architecture of the self-organizing map (SOM). The input vector $\mathbf{x}$ is connected with every neuron (i) in the map through input vector $\mathbf{w}_i$. The winner for the current output (i^*) is shown as well as neighborhood of the winner. Color intensity shows the weights update's strength for the current input.

Let N be the number of neurons in the map, let n be the input vector size. Let weights of a neuron on i-th position (i-th neuron) be denoted $\mathbf{w}_i$. The first part of the algorithm (Alg. 1.1, line 3) is computed using distance calculation between input vector and weight vectors of all neurons in the map. We used squared Euclidean distance (Eq. 1.4).

Algorithm 1.1 Pseudocode of the SOM learning algorithm. Learning ends when defined number of epochs is achieved.

```
for all epochs do
    for all inputs do
        Compute distances between input vector and weights of all neurons
        in the map;
        [competition] Compute best matching neuron (winner);
        [cooperation] Adapt the weights of all neurons;
        Adapt parameters;
    end for
end for
```

$$d(\mathbf{x}, \mathbf{w}_i) = \|\mathbf{x} - \mathbf{w}_i\|^2 = \sum_{j=1}^{n} (x_j - w_{ij})^2 \tag{1.4}$$

The neuron with the smallest distance between its weight vector and the input vector becomes the best matching unit (winner) for the particular input (Eq. 1.5).

$$i^* = \text{arg min}_{i \in N} d(\mathbf{x}, \mathbf{w}_i) \tag{1.5}$$

In order to have similar inputs mapped onto close neurons in map not only the winner but also neighborhood neurons have to be updated. This action will pull the weights of these neurons closer together providing more similar weight vectors and therefore similar inputs are mapped onto close neurons within the map.

Let $dist(i^*, j)$ denote the distance in the grid between the best matching unit (i^*) and another neuron (j). This distance is computed as Euclidean distance (Eq. 1.6) between positions of the neurons in grid (denoted as $\mathbf{pos}(i^*)$ and $\mathbf{pos}(j)$).

$$dist(i^*, j) = \|\mathbf{pos}(i^*) - \mathbf{pos}(j)\| \tag{1.6}$$

Function $h(i, j)$ provides proximity to the winning neuron and is called neighborhood function. It can be computed as the Gaussian function[3] based on distance of neurons in the grid (Eq. 1.7). Parameter σ influences the size of the neighborhood and can decrease through time to achieve more stable results.

[3] The most used types of the neighborhood functions are bubble and gaussian

$$h(i^*, j) = \exp(-\frac{dist(i^*, j)^2}{2\sigma^2(t)}) \tag{1.7}$$

The next step is to change the weights of the neurons in the map. The change of weight (Eq. 1.8) is computed using the neighborhood function (Eq. 1.7) and the distance between the input vector and the weight vector (Eq. 1.4).

$$\Delta \mathbf{w}_j = \eta(t)h(i^*, j)(t)(\mathbf{x} - \mathbf{w}_j) \tag{1.8}$$

Parameter η denotes the learning speed and can also decrease in time to allow better fine-tuning of the weights.

The space complexity of the SOM algorithm is based on the map size (N) and the input weight vector size (n) as the space occupied by SOM is in its weights. To save (and restore) SOM its weights and parameters need to be saved. There are only a few parameters (number of rows, columns, learning speed, neighborhood size, etc.) whose count is a constant. Eq. 1.9 shows the space complexity of SOM.

$$S(\text{SOM}) = O(Nn + c) = O(Nn) \tag{1.9}$$

The time complexity is computed from the learning algorithm and the learning steps. Some steps are constant in time and are not influenced by the data set size. Let p denote number of epochs and m denote data set size. The first step, setting input, costs the size of the input vector. The second step, computing differences between input and weight vectors, costs computing squared Euclidean distance for every neuron in the map. The time complexity of squared Euclidean distance (Eq. 1.4) is $O(n^2)$ and is computed for every neuron in the map. The step after computing is finding winner which means finding lowest value of the previously computed distance. The next step, adapting weights (Eq. 1.8), costs computing in constant time (η, $h(i^*, j)$) (map dimensionality is a constant of two), squared Euclidean distance and is computed for every neuron in the map ($O(cn^2)$). The last step, adapting parameters, is a constant. Together the resulting time complexity is summarized as the time complexity of the input:

$$T(\text{SOM}_{input}) = O(n + Nn^2 + N + Ncn^2 + c) = O(Nn^2) \tag{1.10}$$

The time complexity of one epoch is m-times higher as the input time complexity. The whole learning algorithm's time complexity is p-times higher than that of one epoch:

$$T(\text{SOM}) = O(pT(\text{SOM}_{epoch})) = O(pmT(\text{SOM}_{input})) = O(pmNn^2) \tag{1.11}$$

From the equation follows that the time complexity increases linearly (sequentially) with increasing number of epochs, data set size and the map size and polynomially (quadratically) with increasing the input vector size.

Model SOM is used in feature mapping, vector quantization, dimension reduction, topographic mapping and data mining.

Chapter 2

Artificial neural networks for sequential data

Supervised recurrent neural networks have become an established approach for processing sequential data, mainly based on the next-item prediction paradigm, e.g. in sentence processing, or time-series prediction (Gori et al., 1997). These models have naturally been generalized to processing more complex data structures such as trees or directed acyclic graphs (Frasconi et al., 1998; Sperduti and Starita, 1997) [1]. The training method for recursive networks is a straightforward generalization of standard backpropagation through time (Sperduti and Starita, 1997). Moreover, important theoretical investigations from the field of feedforward and recurrent neural networks have been transferred to recursive networks (Frasconi et al., 2001; Hammer, 2000). It is also possible to use the standard backpropagation when dealing with n-ary trees, as shown e.g. in case of the recursive auto-associative memory (Pollack, 1990) and its numerous variations.

There are some unsupervised neural network models that can be used for structure processing as well. Simple models without recurrency can not be used as they have no means to capture the structure of the input. Only the content is learned without connection to the structure. Therefore we are working with recurrent models: recurrent self-organizing maps (SOM) and auto-association models (i.e. RAAM). There are multiple differences between these models:

- some can decode as well as encode information
- some retain input data closeness

[1] We will refer to recurrent models with respect to processing sequences, and to recursive model with respect to processing trees.

- recurrent binding saves different amount of information
- different learning algorithms are used

In this chapter we will review the existing models that are used on the basic type of structured data – sequences. In the next chapter we will present enhancements to the existing models to enable processing of more complex structured data types – trees. We will firstly briefly mention supervised models. Then we will focus in more detail on the unsupervised models.

2.1 Supervised recurrent neural networks

Supervised recurrent neural networks are well established approach and are used frequently for the tasks such as to predict or to generate the next-item. The presented models are simple recurrent networks and echo state networks.

2.1.1 Simple recurrent networks

For the structured data to be processed the simplest approach is to add memory, place where information about the previous inputs can be stored. In the case of a simple recurrent networks (SRN) (Elman, 1990) this role is taken by an additional layer (Fig. 2.1).

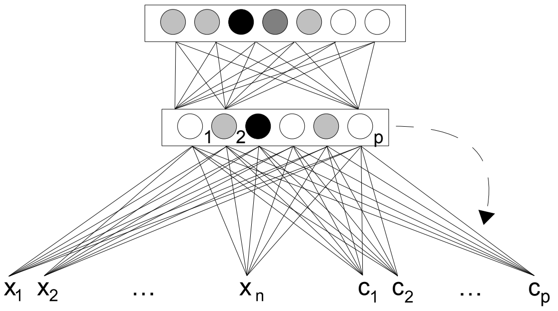

Figure 2.1: Architecture of Elman network (SRN). The second input (c_i) is a time delayed activation of the hidden layer called context.

To create SRN from multi-layered feedforward network the context layer is needed. The context layer size is the same as the size of the layer it copies from (hidden layer or output layer). The context layer is updated by copying the neuron activation (1 to 1) from the chosen layer. In case of Elman network (Elman, 1990) the context layer copies values from the hidden layer. There are multiple SRN models with different types of the context layers and also with more than one context layer (Sinčák and Andrejková, 1996a,b).

Learning algorithms used for SRNs are gradient learning algorithms that use error correction. These can be differentiated into two types: online and batch learning algorithms. The former type concentrates on updating weights after every presentation of the vector from the input from the data set. In this case the weights are updated as often as the information about the size of the change is known. Therefore the change is usually small and continuous. Among the online learning algorithms used for simple recurrent networks are standard back-propagation (BP), back-propagation through time (BPTT) online (Werbos, 1990) and real time recurrent learning (RTRL) (Williams and Zipser, 1989).

The quicker batch learning algorithms use the idea of the accumulation of the changes throughout the presentations of the data set. That means weights are constant during the presentation of the input and are updated only after the whole input was presented. This is faster as the weight update costs a measurable amount of time (it depends on the input size). The batch learning algorithms, i.e. epoch-wise back-propagation through time, can be used if the online learning is not required.

Both learning algorithm types belong to the learning with teacher. To learn the information in this type of learning the desired output is needed. The difference between the desired and the real output drives the change of weights in every learning algorithm of this type.

2.1.2 Echo state networks

Another approach is to create reservoir with a typically much higher number of neurons which provide rich dynamics. The connections to the hidden layer are untrained and its activation is afterwards mapped to desired outputs. This model was proposed by (Jaeger, 2001) and it is called echo state network (ESN). Main differences, compared to SRN, are: inputs can be connected directly with the output layer and only output weights are updated during training. For proper functionality of ESN must reservoir fulfill echo state property. It can be achieved in following way: recurrent weights between neurons in the reservoir have to be initialized in such a way that spectral radius of a weight matrix is less than one.

Echo state networks simplify training procedure of SRN. Only one layer of weights is updated what allows usage of linear regression algorithm instead of complex gradient training methods.

Using ESN model has its advantages and disadvantages. Easier and faster learning compared to the SRN model led to the successful applications of the

ESN models in various fields of signal processing (Jaeger, 2001; Lukosevicius and Jaeger, 2009).

As it turns out, Markovian property of the reservoir (resulting from its spectral radius) is sufficient for various applications. At the same time, Čerňanský and Tiňo (2007) showed by simulations that performance of ESN is equal to that of variable length Markov models (VLMM). However the SRN model, when properly trained can exceed both ESN and VLMM in the next-item prediction task in case of symbolic sequences.

2.2 Recurrent self-organizing maps

From the models using learning without teacher, self-organizing maps can also be extended into processing sequences. The class of such models is called recurrent SOM. Recurrent SOM models use time delayed binding to remember previous inputs. They differ in how the time delayed binding is added. The simplest model uses neurons as leaky integrators and the most complex uses copy of the activations of the whole map from the previous time step.

2.3 Temporal Kohonen map

The first (very simple) way of adding recurrency to an existing SOM model is to use leaky integrators, i.e. recurrent binding is used by every neuron but using it only for itself (self loop). This type of recurrent self-organizing neural network uses temporal Kohonen map (TKM). In the TKM model (Chappel and Taylor, 1993) activation of neuron is computed differently than SOM model (Eq. 2.1).

$$V_i(t) = \tau V_i(t-1) - \frac{1}{2}\|\mathbf{x} - \mathbf{w}_i\|^2 \tag{2.1}$$

In the equation $0 < \tau < 1$ is a time constant, $V_i(t)$ is the activation of the unit i at time step t while $\mathbf{w}_i$ is the weight vector of the unit i and $\mathbf{x}$ is the input pattern. In this case the unit with the maximum activity is the winner.

The architecture of the TKM model is the same as the SOM model (no neurons or layers are added). This model is efficient (no extra computation) but lacks expresivity of the more complex recurrent models. Some problems of this model were shown (Koskela et al., 1998a), it appears that it may be possible to properly train a TKM only for a relatively simple input spaces.

As a consequence Recurrent SOM model was proposed as a modification of TKM model.

Compared with SOM the space and the time complexity remains the same for this model.

2.4 Recurrent SOM

Very similar model to the TKM is the Recurrent SOM (RSOM). RSOM (Koskela et al., 1998b) uses neurons as leaky integrators as well (the same type of recurrent binding as in TKM). The difference is in the type of information that flows through the recurrent connection. The architecture of the RSOM model is the same as of the SOM model.

In this model final distance is computed by merging the distance between input and weight vectors and also the previous distance (recurrent binding) (Eq. 2.2). The winner is computed differently as the resulting distance is a vector (Eq. 2.3)

$$\mathbf{d}_i^{'}(t) = \alpha(\mathbf{x} - \mathbf{w}_i) + \beta \mathbf{d}_i^{'}(t-1). \tag{2.2}$$

$$i^* = \arg\ \min_{i \in N} \left\| \mathbf{d}_i^{'}(t) \right\| \tag{2.3}$$

Parameter α is used to influence strength of the recurrent binding, parameter β is usually set to $1 - \alpha$. The recurrent binding transfers information about how well neuron performed in the previous time step. This causes map to prefer neurons that are winning more often. Large α corresponds to short memory while small values of α (large values of β) correspond to long memory and slow decay of activation.

Also in RSOM model the space complexity remains the same as for SOM model. The time complexity is the same as well.

2.5 Merge SOM

Merge SOM (MSOM) was developed by Strickert and Hammer (2004). This model is based on classic SOM using context vector for information about the previous winner. MSOM algorithm can also be used in models without the grid (for example neural gas). MSOM uses context and input weights of the previous winner. Both vectors are then merged together. Visualization of Merge SOM's architecture can be seen on Fig. 2.2.

Context vector is of the same length as the input vector. It consists of merged data from the winner – its context weights and its input weights.

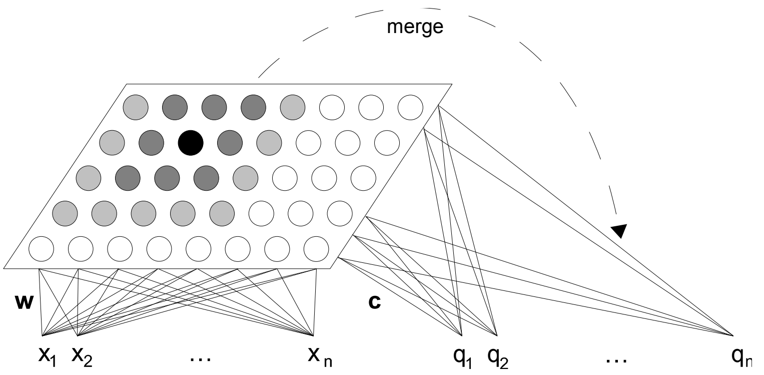

Figure 2.2: Architecture of Merge SOM (MSOM). Context layer consists of merged data from the winner (**q**) in the previous time step. Context weights connect context with every neuron in the map.

In the first phase (competition) winner is calculated using both input and context:

$$d_i = \alpha\|\mathbf{x} - \mathbf{w}_i\|^2 + \beta\|\mathbf{q} - \mathbf{c}_i\|^2, \tag{2.4}$$

where **q** is the so-called context descriptor.

Final distance is dependent on both input and context weights. Parameters α and β (usually $\beta = 1 - \alpha$) are used to balance the context influence on winner's calculation.

The second phase (cooperation) consists of updating both input weights and context weights. The input weights are updated just as in SOM model (1.8). The context weights are updated similarly (Eq. 2.5).

$$\Delta\mathbf{c}_j = \eta(t)h(i^*, j)(t)(\mathbf{q} - \mathbf{c}_j) \tag{2.5}$$

The context has to be updated as well. As mentioned earlier the context is computed by merging previous winner's context and the input weights. To balance the influence of both weights parameter γ is used (Eq. 2.6).

$$\mathbf{q} = \gamma\mathbf{w}_{i^*} + (1 - \gamma)\mathbf{c}_{i^*}, \tag{2.6}$$

Both input and context weights in the equation are taken from the current winner. That means that only the information about the winner is merged into the new context. By merging the weight vectors this model loses also part of the information about the winner.

For example we can take sample sequence 'abc' and train the MSOM model. First input will consist of encoded symbol 'a' and empty context (Fig. 2.3). Let winner for this input be in position [1, 3].

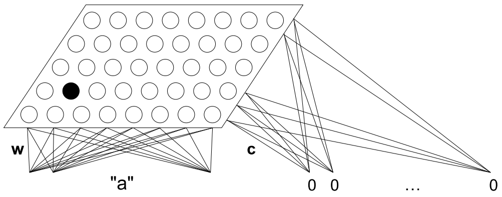

Figure 2.3: MSOM with input ‘a’ and without the last winner (first input in the sequence).

The next input will consist of encoded ‘b’ as input and merged context of neuron [1, 3] which was a winner for the input ‘a’ (Fig 2.4).

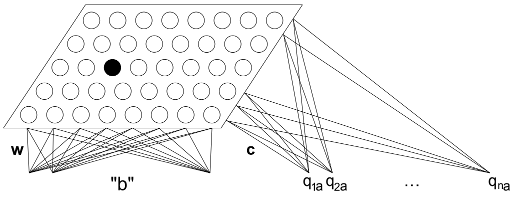

Figure 2.4: MSOM with input ‘b’ and the merged context of the last winner.

The winner of this input can be for example at position [2, 2] so the last input will look like this: encoded ‘c’ and merged context of neuron [2, 2] (Fig 2.5). The new winner can be located at position [5, 3].

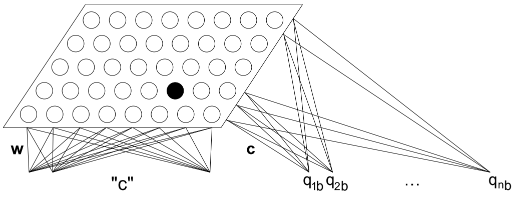

Figure 2.5: MSOM with input ‘c’ and the merged context of the last winner.

This means that neuron at position [5, 3] encodes the sequence ‘abc’. Also the neuron at position [1, 3] encodes the sequence ‘a’ and the neuron at position [2, 2] encodes the sequence ‘ab’. That does not necessarily mean that these neurons encode only these sequences.

This model uses only data (not position) to update context. Information from non-winning neurons is lost in MSOM.

The space complexity is higher than for SOM as the context weights need to saved for every neuron. But it is asymptotically the same as for SOM model:

$$S(\text{MSOM}) = O(Nn + Nn + c) = O(Nn) \tag{2.7}$$

The time complexity of MSOM is also higher. The difference is in presenting input (context needs to be presented as well), in computing differences between weights and inputs (context weights are added to the equation) and in adapting weights (context weights are adapted as well). The finding winner and adapting parameters steps have the same time complexity as SOM. One new step is update of the context (Eq. 2.6) which costs time n. Also the time complexity is asymptotically the same compared with SOM, therefore:

$$T(\mathrm{MSOM}_{input}) = O(2n + 2Nn^2 + N + 2n^2 + c + n) = O(Nn^2) \tag{2.8}$$

The time complexity of the epoch and therefore of the whole learning has to take the size of the structure into account:

$$T(\mathrm{MSOM}) = O(pT(\mathrm{MSOM}_{epoch})) = O(pmM_{\max}T(\mathrm{MSOM}_{input}))$$

$$T(\mathrm{MSOM}) = O(pmM_{\max}Nn^2) \tag{2.9}$$

where $M_{\max}$ is the maximum value of M (structured input size), in case of sequences it is the longest sequence's length.

2.6 Recursive SOM

The most complex model presented is recursive SOM (RecSOM) (Voegtlin, 2002a). This recurrent model's context is based on copying activities of the whole map (Fig. 2.6).

Learning in RecSOM starts as with classic SOM. Input is presented and the winner is computed from distance between inputs and weights. Context inputs are added to the winner computation as well:

$$d_i = \alpha\|\mathbf{x} - \mathbf{w}_i\|^2 + \beta\|\mathbf{y} - \mathbf{c}_i\|^2, \tag{2.10}$$

where vector $\mathbf{y}$ represents previous activations of the map. In order to get stable representations Voegtlin proposed function (Eq. 2.11) that limits the values to the interval $(0, 1)$.

$$y_i = \exp(-d_i) \tag{2.11}$$

After winner computing all weights (both input and context) are updated. The input weights use SOM update algorithm (Eq. 1.8) and the context weights are updated according to the rule:

$$\Delta\mathbf{c}_j = \eta(t)h(i^*, j)(t)(\mathbf{y} - \mathbf{c}_j) \tag{2.12}$$

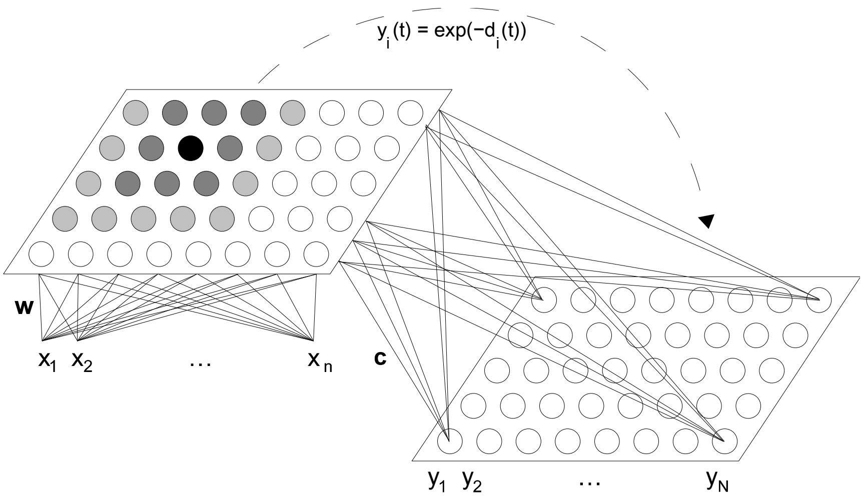

Figure 2.6: Recursive SOM model (RecSOM). The context layer of RecSOM consists of the activation of the whole map ($\mathbf{y}$) in the previous time step.

The complexity of the RecSOM model lies in the computation of the distance and in the update of all weights in the map. This depends on the size of the context weight vector which, in the case of RecSOM, is of size N.

RecSOM's space complexity is significantly higher than for SOM. Every neuron in the map has recurrent connection with all the activations that adds up to the space complexity of the model. Usually the size of the map is bigger than input vector size and therefore the resulting space complexity follows:

$$S(\text{RecSOM}) = O(Nn + NN + c) =_{n \leq N} O(N^2) \tag{2.13}$$

For the time complexity similar thought applies. The first, the second and the fourth step needs to be updated. The new step, context update, is computed using Eq. 2.11 for every neuron in the map:

$$T(\text{RecSOM}_{input}) = O((n + N) + (Nn^2 + NN^2) + N + (n^2 + N^2) + c + N)$$

$$T(\text{RecSOM}_{input}) =_{n \leq N} O(N^3) \tag{2.14}$$

RecSOM and MSOM comparison shows that RecSOM has higher values of both space and time complexity.

The last two models were compared (Farkaš and Vančo, 2007a) on four different data sets with various difficulty. The results showed the topography of the trained maps based on sequence suffixes. The state space organization had usually Markovian dynamics which follows from existence of fixed-point attractors in the state space with asymptotic dynamics. RecSOM was able to generate non-Markovian dynamics which means, in theory, unlimited memory depth. However, when non-Markovian dynamics is not required MSOM model is a more effective replacement for RecSOM as its time and space complexity is lower.

Neubauer (2005) proved that RecSOMs with a slightly modified (i.e. normalized) context can simulate finite state automata. RecSOMs with a simplified context (i.e. only the maximum values remain, the exponential function is substituted by a semilinear function) can simulate pushdown automata.

Even though the weight settings required for the simulations cannot be learned by standard training and simulations with the original context function could not be constructed for the general case, the results obtained may serve as theoretical arguments in favor of recurrent (and also recursive) SOM architectures and their potential.

2.7 Gradient learning approximation

One of the simplest and most intuitive learning paradigms in self-organizing maps is Hebbian learning. However learning process can be explicitly formulated in terms of a cost function which must be optimized and thus create alternatives to Hebbian learning. Most unsupervised Hebbian learning algorithms for simple vectors can be interpreted as a stochastic gradient descent on an appropriate cost function. However, the learning for structured data cannot be interpreted as an exact gradient descent method.

Therefore, Hebbian learning can be seen as an efficient approximation of the precise stochastic gradient descent. Error functions of RTRL learning can be defined and also gradient descent on the error functions of another standard algorithm, backpropagation through time or structure, can be formulated.

Hammer et al. (2004a) formulated the cost functions for the general framework and derived two ways to precisely compute the gradients. Hebbian learning is much less costly than the precise approaches as it disregards the contribution of substructures. Also, supervised training mechanisms can be recovered within the general framework. Choosing the appropriate cost function leads to standard training algorithms for supervised recurrent networks like BPTT and RTRL.

2.8 RAAM

Recursive Auto-Associative Memory (RAAM) (Pollack, 1990) model is a model learned using self-supervision. The learning of this model is based on auto-association, i.e. the input is the same as required output of the model. This model, with different modifications, has been the most used model since 1990.

RAAM consists of two logical parts: encoding and decoding. These parts are connected and are trained together using auto-association. It consists of three physical parts: input, hidden and output layers. The input and the output layers have the same length (twice the input vector size for binary RAAM) and the hidden layer is smaller to compress inputs, it has the length of the input vector (Fig. 2.7).

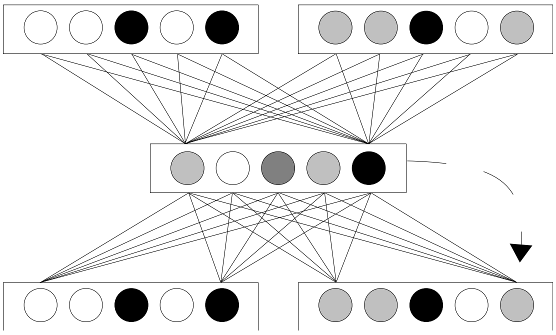

Figure 2.7: Binary RAAM. Inputs are compressed into the hidden layer and then decompressed into the output layer. As auto-associator, the inputs are the desired outputs.

The training of the model for sequences proceeds in a reverse topological order, i.e. the last parts of the sequence are presented first. The last two parts are presented, outputs are computed and the weights are changed using the standard backpropagation algorithm using the presented parts of the input as the desired output. The next part of the input is presented on the left side of the input vector while on the right side of the input copy of hidden layer's previous activation is presented. Again, the auto-association is used to get the desired output. The whole sequence is presented from the last part to the first. After the first part has been presented the hidden layer's activation encodes the whole sequence.

After the training, RAAM can also decode the activation of the hidden layer back to its original structure. The decoding is based on decision when the output is already terminal part of the input or the compressed activation of some substructure.

This model was created to process tree structures. For binary trees the same model is used. Again, input is presented in reversed topological order, from leaves to the root, left child is presented on the left, right child on the right side of the input vector. In case of trees with arity k, this model has k inputs and therefore k outputs. The training is similar to the binary trees case. Inputs (tree vertices or already compressed representations) are presented and trained using auto-association.

In contrast to recurrent SOM models model RAAM uses auto-associativity to order hidden (compression) layer while recurrent SOM models use similarity principle. These models can be compared with regards to data compression. Similar inputs have similar representations in both model types, where representation in RAAM model is activity of the hidden layer and representation in recurrent SOM models can be either weight vector of the winner or the activity of the map. However these models cannot be compared in data decompression as recurrent SOM models do not have implicit output (decompression) layer. Recurrent SOM models can be seen as alternatives to the RAAM model, their advantage being in visualization of the data, data analysis and multiple results co-existence when creating population of detectors.

Chapter 3

Self-organizing maps for tree data

Tree data is more complicated as the structure is not linear as in the case of sequences. The problem is that for every vertex in the tree its successors (or children) have to be already processed by the map. This need is based on context values that have to be known before the processing of the vertex. Therefore processing of the tree is bottom-up procedure and ends with processing of the tree root. Good practice can be to process the lowest depth first and work up to the root. In the implementation also right to left processing was used. Combining these techniques a procedure was created that processes tree from rightmost element of the bottom, moving left and then processing higher levels until the root is reached. The ordering in which vertices are presented is called topological ordering. All algorithms presented can be also used in processing of oriented acyclic graphs but in this case the bottom-up procedure needs to be redefined so that parent vertices are processed only if all child vertices have been processed already. SOM models processing tree data are called recursive SOM models.

3.1 SOM for structured data

SOM for structured data (SOMSD) (Hagenbuchner et al., 2003) is a model that was created to work on trees. It can be used to work on sequences as well but its primary role is to work on trees and acyclic graphs. SOMSD uses the information about the winner's position in the previous time step as the context vector. That means the context vector size is two in case of two dimensional map. For trees with fan-out k, k contexts are needed to encode the previous state of the map. When working with binary trees the positions of left and right subtree winners are used as contexts (Fig 3.1). $\mathbf{r}$ denotes a vector consisting of two values: winner's position x and y in the grid.

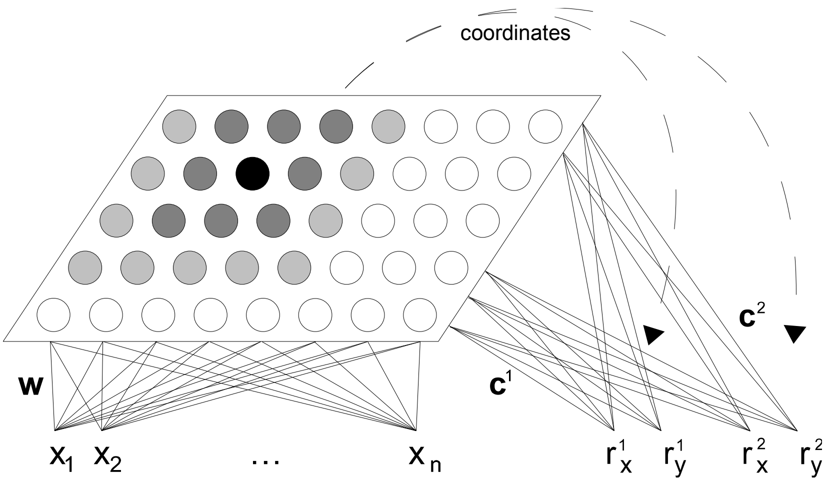

Figure 3.1: Model SOMSD for tree structures. The model adds contexts that consists of the winner coordinates in the previous time step (**r**).

For SOMSD, the distance of unit i from a tree with fan-out k while reading a vertex v is computed as

$$d_i(v) = \alpha\|\mathbf{v} - \mathbf{w}_i\|^2 + \beta(\|\mathbf{r}_{ch(1)} - \mathbf{c}_i^{(1)}\|^2 + \ldots + \|\mathbf{r}_{ch(k)} - \mathbf{c}_i^{(k)}\|^2) \qquad (3.1)$$

where $ch(j)$ denotes the winner index for jth child, $\mathbf{r}_{ch(j)}$ are winner coordinates in the lattice and $\mathbf{c}_i^{(j)}$ is ith context vector of jth child. Hence, the context vector (for each child) is only two-dimensional (we assume 2D maps) and has integer components. In Eq. 3.1 (and also in subsequent equations), parameters $\alpha > 0$ and $\beta > 0$ respectively influence the effect of the current input and all contexts on the neuron's profile.

New context is computed as the x and y position of the winner and saved for parent computation.

As mentioned earlier training follows the tree structure. The first inputs presented is the most bottom layer – the leaves. Following are upper levels of the tree until the root of the tree is reached. In the example simple binary tree over 3 symbols (Fig. 3.2) is used as input.

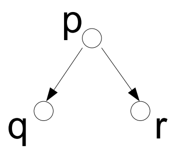

Figure 3.2: Basic tree structure. Can be encoded as linear version: (p(qr))

The first input is the rightmost leaf ‘r’. As leaves have no subtrees the position of winners for both left and right winners is [-1, -1]. The winner of this input can be at position [6, 2]. The next input is the second leaf ‘q’ with the same position of previous winners. This time winner neuron can be at position [3, 1]. The last input is ‘p’ with the both left and right subtrees. For this input the winner can be at position [7, 0]. As we know the winner positions, we can use them as a part of the input (Fig. 3.3).

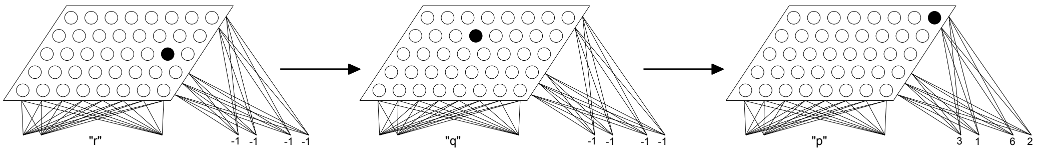

Figure 3.3: Computation of tree structure with SOMSD model

To compute space and time complexity the arity of the tree is needed (k). SOMSD requires k context layers and one input layer, i.e. every neuron in the map needs $(n + 2k)$ weights. Therefore the space complexity is:

$$S(\text{SOMSD}) = O((n + 2k)N + c) = O(N(n + k)) \tag{3.2}$$

The time complexity is divided in steps as before: the finding winner and adapting parameters steps have the same time complexity as SOM. Presenting input (with contexts) costs $O(n+2k)$. The computation of difference between the weights and the input with contexts costs (Eq. 3.1) $O(N(n^2+k))$. More weights need to be updated as well, the cost is $O(N(n^2 + k2^2))$. And the new context needs to be computed $O(c)$. The sum of all steps:

$$T(\text{SOMSD}_{tree-input}) = O((n + 2k) + N(n^2 + k) + N + N(n^2 + 4k) + c + c)$$

$$T(\text{SOMSD}_{tree-input}) = O(N(n^2 + k)) \tag{3.3}$$

The time complexity of the epoch and the whole learning use the same thought as in Eq. 2.9:

$$T(\text{SOMSD}_{tree}) = O(pmM_{\max}N(n^2 + k)) \tag{3.4}$$

Hagenbuchner et al. (2005a) proposed extension to SOMSD model called contextual SOMSD (CSOMSD). This extension enables the model to achieve contextual processing of information, i.e. the model is trained using information not only about successors but also about predecessors. One single CSOMSD can replace layers of self organizing maps. Some experiments were conducted to compare existing SOMSD model with CSOMSD (Hagenbuchner et al., 2005b).

3.2 MSOM

In case of tree structures is the situation more difficult than in SOMSD model. Merge SOM can be modified to accommodate tree data (Fig. 3.4).

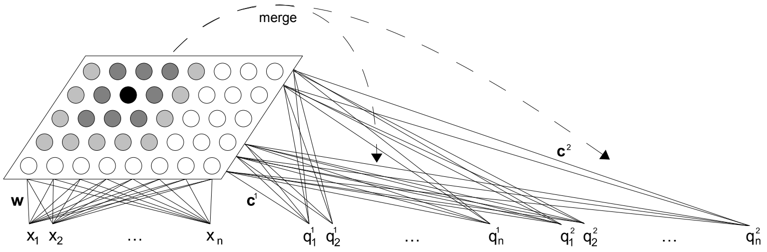

Figure 3.4: MSOM modification for binary tree data. Left context represent left child, right context represents right child.

Similar to the Eq. 2.4 the unit's distance is computed as:

$$d_i(v) = \alpha\|\mathbf{v} - \mathbf{w}_i\|^2 + \beta(\|\mathbf{q}_{ch(1)} - \mathbf{c}_i^{(1)}\|^2 + \ldots + \|\mathbf{q}_{ch(k)} - \mathbf{c}_i^{(k)}\|^2) \tag{3.5}$$

where $\mathbf{q}_{ch(j)}$ is the so-called context descriptor of jth child computed as

$$\mathbf{q}_{ch(j)} = (1-\gamma)\mathbf{w}_{ch(j)} + \gamma(\frac{1}{k}\mathbf{c}^{(1)}_{ch(j)} + \ldots + \frac{1}{k}\mathbf{c}^{(k)}_{ch(j)}) \tag{3.6}$$

Again, the dimensionality of the context vector equals input dimension n.

The space complexity adds k context weights instead of one (Eq. 3.7) to the result:

$$S(\text{MSOM}) = O(Nn + kNn + c) = O(kNn) \tag{3.7}$$

The number of contexts influence the time complexity of the model as well. The original equation (Eq. 2.8) has to be changed to accommodate for k contexts. The first step consists of presenting k contexts of size n and one input and costs $O((k+1)n)$, the difference is computed for all contexts and input as well, it costs $O((k+1)Nn^2)$. Finding winner costs the same amount as in SOM. Then all weights need to be updated which costs $O((k+1)Nn^2)$, adapting parameters is a constant and updating contexts costs $O(kn)$. The resulting time complexity is:

$$T(\text{MSOM}_{tree-input}) = O((k+1)n + (k+1)Nn^2 + N + (k+1)Nn^2 + c + kn)$$

$$T(\text{MSOM}_{tree-input}) = O(kNn^2) \tag{3.8}$$

The time complexity of the epoch use the thought mentioned in SOMSD model (Eq. 2.9 and Eq. 3.4)

$$T(\text{MSOM}_{tree}) = O(pmM_{\max}kNn^2) \tag{3.9}$$

There is one problem with this modification – the sensitivity to data orientation. As a consequence, only certain tree structures could be faithfully represented in this model. Hammer et al. (2004b) argued that MSOM does not distinguish between the branches of a tree, because the operation of unit distance calculation "is commutative with respect to the children. Two trees resulting from each other by a permutation of the vertex labels at equal height have the same winner and the same internal representation" (p. 7). The proposed alternative solutions to compensate for the problem were:

1. The first alternative relies on the encoding of tree structures in prefix notation in which the single letters are stored as consecutive digits of a real number. A specified digit denotes an empty vertex. For example, if vertices are labeled 1,2,3 and empty vertex is labeled 9, the sequence 12993299199 represents the tree 1(2,3(2,1)). This yields a unique representation. The problem here lies in the depth of tree that has to be known before processing to set the length of the input vector.

2. Trees can be represented as real values of which consecutive digits correspond to the single entries. In such a setting, merging of context would correspond to a concatenation of the single representations, i.e. a label $l(v)$ and representation strings s_1 and s_2 for the two subtrees would result in the prefix representation $l(v)s_1s_2$ of the entire tree. Concatenation is not commutative operation and therefore it can be used in case of oriented data. This approach is not very efficient in practice (numbers are easier processed than strings).

However, we show that MSOM can in fact distinguish between branches of a tree and therefore the alternative solutions are not necessary.

3.2.1 Distinguishing branches of a tree

Here we show that although commutativity introduces some loss of information, it does not prevent MSOM from being able to distinguish two trees with permuted branches. (How much information is actually lost has to be further analyzed). We illustrate the argument using a simplest pair of two binary trees T1: (a(b c)) and T2: (a(c b)) that are relevant for the above commutativity condition. These trees are identical regarding the structure but different regarding the content. Let $R^{T1}_{ch_1(a)}$[1] and $R^{T1}_{ch_2(a)}$ be the computed context representations of the two children (labeled b and c respectively)

[1] $R^{T1}_{ch_1(a)}$ denotes the representation of the first child of the node "a" of the tree "T1"

for T1, and let $R^{T2}_{ch_2(a)}$ and $R^{T2}_{ch_1(a)}$ be the context representations of the two children (c and b) for T2. Since they are leaves, it holds that

$$R^{T1}_{ch_1(a)} = R^{T2}_{ch_2(a)} \qquad \text{and} \qquad R^{T1}_{ch_2(a)} = R^{T2}_{ch_1(a)} \quad . \tag{3.10}$$

Let the winner for the first tree be $w1 = \arg\min_i\{d^{T1}_i\}$ where

$$d^{T1}_{w1} = \alpha d(\mathbf{w}_{w1}, '\mathrm{a}') + \beta(d(\mathbf{c}^{(1)}_{w1}, R^{T1}_{ch_1(a)}) + d(\mathbf{c}^{(2)}_{w1}, R^{T1}_{ch_2(a)}))$$

and for the second tree $w2 = \arg\min_i\{d^{T2}_i\}$ where

$$d^{T2}_{w2} = \alpha d(\mathbf{w}_{w2}, '\mathrm{a}') + \beta(d(\mathbf{c}^{(1)}_{w2}, R^{T2}_{ch_1(a)}) + d(\mathbf{c}^{(2)}_{w2}, R^{T2}_{ch_2(a)}))$$

with d denoting the similarity measure (e.g. a Euclidean metric). We show by contradiction that in general

$$d(\mathbf{c}^{(1)}_{w1}, R^{T1}_{ch_1(a)}) + d(\mathbf{c}^{(2)}_{w1}, R^{T1}_{ch_2(a)}) \neq d(\mathbf{c}^{(1)}_{w2}, R^{T2}_{ch_1(a)}) + d(\mathbf{c}^{(2)}_{w2}, R^{T2}_{ch_2(a)}) \tag{3.11}$$

in which case the extended MSOM is able to differentiate between T1 and T2. Let us assume that equality in Eq. 3.11 holds. Using Eq. 3.10 we get

$$d(\mathbf{c}^{(1)}_{w}, R^{T1}_{ch_1(a)}) - d(\mathbf{c}^{(2)}_{w}, R^{T1}_{ch_1(a)}) = d(\mathbf{c}^{(1)}_{w}, R^{T1}_{ch_2(a)}) - d(\mathbf{c}^{(2)}_{w}, R^{T1}_{ch_2(a)})$$

This equality holds if and only if vectors $R^{T1}_{ch_1(a)}$ and $R^{T1}_{ch_2(a)}$ lie on the same (n-1)-dimensional manifold (where n is the context vector size). However, the space of solutions is n-dimensional so only a fraction of them satisfies the above equation. [2] Therefore, there exist only a few cases when commutativity holds and when the two children would be indistinguishable. Hence, this provides extension (cf. Theorem 3 in Hammer et al. (2004b)) of MSOM to the processing of trees.

To verify the validity of our argument, we performed simple simulations in which MSOM was observed to distinguish all binary trees (of depth one) with permuted children using three symbols.

This argument holds even if (for some reason) R is computed as $R_{ch(j)} = \mathbf{w}_{ch(j)}$, i.e. only using winner's input weights, but not context weights. This does not change our argument, because Eqs. 3.10 and 3.11 remain unchanged as does the conclusion.

At the same time, although MSOM can differentiate between permuted branches of a tree, some loss of information occurs due to two reasons. First,

[2] The similar situation also arises in a standard SOM when two different inputs yield the equal distance from the same unit if they lie on the surface of the (n-1)-dimensional hypersphere, centered around that unit's weight vector.

only the information about the winner is saved (as in SOMSD, albeit with a different content), and second, due to the commutativity of operation (the last term in Eq. 2.6). Therefore, MSOM in principle cannot learn all types of structured data. On the other hand, this information loss means faster computation that in most cases is more important than saving every detail in the context.

To verify the validity of our argument, we performed simple simulations in which MSOM was observed to distinguish all the binary trees (of depth one) with permuted children using three symbols.

Simulation

For the simulation we chose data set of all binary trees of depth one (root and two child vertices) over three symbols. The symbols can be visualized as ‘a’, ‘b’ and ‘c’. There are six trees in the data set, two of each symbol as a root. For this very small data set we chose 10×10 map trained 2000 epochs.

The (α, β) parameter space was searched systematically (interval 0–1 with a step 0.1) to confirm the results. The context descriptor parameter was set to its default value $\gamma = 0.5$. The neighborhood size was set to σ:3→0.5 over 1200 epochs and then remained constant. The learning rate was set μ:0.3→0.15 over 1200 epochs, and further decreased linearly down to 0.1.

Symbols were encoded into input vector of size one. Both context vectors had the same size. The value of the input vector was set to 0.1 for ‘a’, 0.5 for ‘b’ and 0.9 for ‘c’.

In all experiments the results were the same, all six trees (and also all three leaves) were separated from each other. So there were nine active neurons (out of one hundred) in the map. The difference between results is in the distance between the active neurons corresponding to similar trees. If β was higher than α the distances were more pronounced as the parameter β strengthens the weight of the contexts.

Tab. 3.1 shows example of one particular trained map (map size 10×10, $\alpha = 0.2$, $\beta = 1.0$).

The position of the winners shows that MSOM is sensitive to data orientation and differentiates between child subtrees.

3.3 RecSOM

The same way as previous models this model can also be extended to process trees with arity k by adding k contexts. Equations are updated to work with

input	**winner**	**winner position**
a	9	[0,9]
b	59	[5,9]
c	99	[9,9]
(a(bc))	2	[0,2]
(a(cb))	20	[2,0]
(b(ac))	0	[0,0]
(b(ca))	40	[4,0]
(c(ab))	90	[9,0]
(c(ba))	70	[7,0]

Table 3.1: All inputs and their corresponding winners and winner positions for one particular trained map.

k contexts and parameters α and β control the contexts' and the input's influence of the distance.

In case of trees unit's distance in RecSOM is computed as

$$d_i(v) = \alpha\|\mathbf{v} - \mathbf{w}_i\|^2 + \beta(\|\mathbf{y}_{ch(1)} - \mathbf{c}_i^{(1)}\|^2 + \ldots + \|\mathbf{y}_{ch(k)} - \mathbf{c}_i^{(k)}\|^2) \qquad (3.12)$$

where the components of the context vector $\mathbf{y} = [y_1, y_2, ..., y_N]$ are computed using Eq. 2.11.

The dimensionality N of the context vector (for each child) makes this architecture computationally the most expensive regarding both the space and the time complexity.

For the space complexity the computation, the input weights are the same as for the sequence RecSOM ($O(Nn)$). The context weights are in this case multiplied k times for each context ($O(kN^2)$):

$$S(\text{RecSOM}) = O(Nn + kN^2 + c) = O(kN^2) \qquad (3.13)$$

Adding k contexts changes four of six steps in the learning of one input. In the first changed step input and all contexts are presented. In the second step the distance between input weight and input as well as context weights and contexts is computed. The third step is updating all weights and the fourth step is computing new context.

$$T(\text{RecSOM}_{tree-input}) = O((n+kN)+(Nn^2+kN^3)+N+N(n^2+kN^2)+c+N)$$

$$T(RecSOM_{tree-input}) = O(Nn^2 + kN^3) \qquad (3.14)$$

This model has the highest time complexity so the overall time complexity of this model is:

$$T(\text{RecSOM}) = O(pmM_{\max}N(n^2 + kN^2)) \qquad (3.15)$$

where p denotes number of epochs and m denotes the data set size. Still, in this most complex model the time complexity is polynomial and not exponential.

The comparison of all three models (SOMSD, MSOM and RecSOM) in terms of algorithm complexity shows that

$$T(\text{SOMSD}) < T(\text{MSOM}) < T(\text{RecSOM})$$

and

$$S(\text{SOMSD}) < S(\text{MSOM}) < S(\text{RecSOM}).$$

3.4 GSOMSD

Hammer et al. (2004a) introduced unifying framework for these models. GSOMSD, Generalized SOM for Structured Data, uses parameters to cover all models. Only the basic unifying structure of the algorithm is defined.

Algorithm 3.1 Pseudocode of the GSOMSD training algorithm. This code provides general framework for all presented models and was used in implementation of the models.

```
initialize the weights at random
repeat
    choose some training pattern T
    for all subtrees t in T in inverse topological order do
        compute $d_i$ for all neurons
        compute the neuron $i^*$ with greatest similarity
        adapt the weights of all neurons simultaneously
    end for
until end of training
```

All presented models can be described using this unified model. Hammer et al. (2004a) also presents how to initialize GSOMSD to get presented models.

This framework was used in real implementation of all models and all presented experiments were done using this implementation. This can be viewed as a proof of concept for GSOMSD.

Chapter 4

Model comparison

All presented models have different types of feedback and their differences are not clear from the feedback type. Exact comparison is needed to provide basic understanding how these models perform in experiments. This can be used to decide which model is appropriate for the task required. The types of tasks for the models cover data and structure visualization, memory and clustering.

We focus on the three most complex SOM models (SOMSD, MSOM and RecSOM) and their ability to differentiate among the trees. SOMSD uses reference to the winner position in the grid, MSOM refers to the winner content, and RecSOM refers to the whole map activation. [1] Earlier models, such as TKM or RSOM are not included, due to the strict locality of their feedback, which leads to inferior performance of these models. We assess and compare the performance of the three models using six quantitative measures. For testing we use the tree data sets of increasing complexity: binary syntactic trees, ternary linguistic propositions, and 5-ary graphical data of house structures.

4.1 Performance measures

When dealing with recursive SOMs, we bear in mind that the traditional use of these models is for data clustering (be it points, sequences, or trees) and visualization (low-dimensional projection). For clustering, the number

[1] MSOM and RecSOM are not explicitly designed for optimum representation of a fixed lattice structure, so in principle they can be also used in the Neural Gas (NG) model (Martinetz and Schulten, 1991), which is not possible for SOMSD. The dynamic topology of NG would allow more flexibility and would hence lead to better representations if map visualization was not one of the goals.

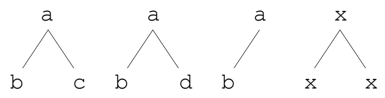

Figure 4.1: Example of two trees (left half) and their overlaps related to TRF and STRF measures respectively (right half).

of map units has to be smaller than the number of inputs in the data set. The inverse property to clustering is the differentiation among inputs, i.e. assigning unique output representations to different inputs. However, we think that a reverse ratio (i.e. more units than inputs) also makes sense (as will be the case of syntactic binary trees), at least for visualization purposes (e.g. in the semantic map (Ritter and Kohonen, 1989)). In addition, in this case we can think of a recursive SOM as performing data transformation (and visualization) from a symbolic space of trees to a vector space (e.g. on a regular lattice). As a consequence, we should employ measures that consider all these aspects. The measures used in our experiments are listed in Table 4.1 and are explained below.

Table 4.1: The quantitative measures used for evaluating the models.

Acronym	Name	Purpose
TQD	Tree Quantizer Depth	content and structural memory
STQD	Structure-only sensitive TQD	structural memory
WD	Winner Differentiation	fraction of unique winners
MED	Minimum Euclidean Distance	between map output activations
QE	Quantization Error	statistics of labels distribution
LWC	Locality-Winner Correlation	spatio-temporal information

First, we introduce two pairs of related numerical measures. The first measure is inspired by the quantizer depth introduced for symbolic sequences for measuring the amount of memory captured by the map (Voegtlin, 2002b). It is defined as the average size of the unit's receptive field, i.e. the common suffix of all sequences for which that neuron becomes the winner. [2] In case of trees, we propose an analogue named Tree Quantizer Depth (TQD), computed as the average size of the Tree Receptive Field (TRF), i.e. the common "suffix" (subtree) of all vertices for which that neuron is the winner. The size

[2] Temporal extensions of SOM that build localist representations of symbolic sequences are essentially context quantizers: the higher the memory depth, the better accuracy in sequence reconstruction.

of TRF is calculated as the number of tree vertices it contains. Hence,

$$\mathrm{TQD} = \sum_{i=1}^{N} p_i s_i \tag{4.1}$$

where p_i is the probability of unit i becoming a winner and s_i is the (integer) size of its TRF. For instance, a unit i winning for just two trees $(\mathsf{a}(\mathsf{bc}))$ and $(\mathsf{a}(\mathsf{bd}))$ will have $s_i = 2$. Weighted averaging is required since some subtrees in the data may be more frequent than the others which should affect the calculation of the model's memory depth. Figure 4.1 illustrates the two trees and their ovelap related to TRF (the second tree from the right).

Unlike sequences, which have linear structure, trees provide room for distinguishing between the content in structure (captured by TQD) and the bare structure. This leads to the second measure, the Structure-only sensitive TQD (STQD), which is less strict, since it is based on structure-only sensitive TRFs (STRFs). STRF considers only the structure of a tree by distinguishing between the leaves and the inner vertices, not between the vertices with different labels (i.e. their content). The measure is defined as

$$\mathrm{STQD} = \sum_{i=1}^{N} p_i s'_i \tag{4.2}$$

where s'_i denotes STRF of ith neuron (i.e. the number of common elements in the subtree while only looking for label/subtree differences) and p_i is the same as above. For instance, a unit i winning for just two trees $(\mathsf{a}(\mathsf{bc}))$ and $(\mathsf{a}(\mathsf{bd}))$ has $s'_i = 3$. Figure 4.1 illustrates the two trees and their overlap related to STRF (rightmost tree).

We introduce the next pair of measures to assess the discrimination capacity of the models to unambiguously represent different trees. This implies the ability to uniquely represent all vertices (subtrees) contained in all trees (Hammer et al., 2004b). One view of looking at the representation of a vertex implies that a map reserves a separate winner for it: The winner differentiation (WD) refers to the level of winners and is computed as the ratio of the number of different winners for the entire data set and the size of the data set. Let $\mathcal{T}$ be the set of all trees and $V(T)$ the set of vertices of a tree T. If $\mathcal{V} = \{v \in V(T), \forall T \in \mathcal{T}\}$ denotes the set of all vertices in the data set, then

$$\mathrm{WD} = \frac{|\{j \mid \exists t : j = i^*(t)\}|}{N_{\mathcal{V}}} \tag{4.3}$$

where $N_{\mathcal{V}}$ denotes the number of all inputs in the data set. WD<1 indicates that not all vertices could be uniquely represented by the map. It is true that

WD depends on the relative number of units with respect to inputs (vertices). Anyway, it holds in general (i.e. regardless of the data set) that the higher WD, the higher proportion of inputs can be represented by distinct winners.

An alternative view at input discrimination is based on a distributed representation of vertices: The corresponding more "detailed" measure (MED) looks at the differences between map output activation vectors. The normalized minimum Euclidean distance is computed as

$$\mathrm{MED} = \min_{k \neq l}\{\|\mathbf{y}(T_k) - \mathbf{y}(T_l)\|/N\}, \tag{4.4}$$

where $\mathbf{y}(T_m)$ is the map output activation vector (whose components are obtained using Eq. 2.11) corresponding to the processing of the root of the tree T_m. If not by WD, the two trees (vertices) have a better chance to be distinguished by MED.

To embrace other aspects of model behavior, we also include quantitative measures defined earlier (Steil et al., 2006). Quantization error (QE) captures information about how well the (input) weight vectors retain statistical information about the labels distribution. [3] We define QE formally as [4]

$$\mathrm{QE} = \frac{1}{N_{\mathcal{V}}} \sum_{T \in \mathcal{T}} \sum_{v \in V(T)} \|\mathbf{x}_v - \mathbf{w}_{i^*}\|^2 \tag{4.5}$$

Hence, QE computes distances in the input space focusing on spatial information and ignoring any structural information contained in the trees. Locality-Winner Correlation (LWC) coefficient was introduced to measure how the spatiotemporal information is encoded into the maps. LWC is defined as a correlation coefficient between "locality" of the trees (being a property of the data set) and the number of distinct winners for the trees, i.e.

$$\mathrm{LWC} = \mathrm{CorrCoef}\{L(T), |win(T)|\} \tag{4.6}$$

Locality of a tree T is defined as: $L(T) = \sum_{v \in V(T)} \sum_{u \in \downarrow v} \|\mathbf{x}_v - \mathbf{u}\|$ where $\downarrow v = \{u | \exists path(v, u)\}$ denotes a list of all child vertices u of vertex v. The number of (distinct) winners for T is defined as

$$win(T) = \left\{ i^* | \exists v \in V(T) : i^* = \arg\min_{i \in N}\{d(\mathbf{x}_v, \mathbf{w}_i)\} \right\}$$

[3] This measure was mainly inspired for the data sets with all labeled vertices, such as our 5-ary graphical data set. For completeness we also report QE for our binary and ternary data sets.

[4] Unlike (Steil et al., 2006) we normalize QE with respect to the number of vertices in the data set to make the results comparable across data sets.

Six measures defined above are used in forthcoming experiments. In all cases, the higher the measure value, the better the performance, except for QE where the opposite is true.

4.2 Experiments

We set the map parameters experimentally according to the model and the task difficulty. We started with basic maps of 10×10 units in case of binary data, and 15×15 units for both ternary propositions and the 5-ary graphical data. In an effort to get a maximum of each model with respect to the above measures, we systematically searched the (α, β) parameter space. Each model can trade-off the effect of inputs and contexts on map formation by these parameters, which were searched experimentally in the interval 0–1 with a step 0.1 for the initial map size with $N = 100$ units. In RecSOM, α was also varied up to 4 (with a step 0.4). [5] In all models, we could observe that varying these two parameters traded-off the effect on leaves (increasing α) and non-leaves (increasing β). More specifically, changing α (while keeping β constant) did not affect output representations of leaves but led to the overall decrease of activations for trees. Increasing β (with constant α) led to gradual vanishing of output representations of leaves, and to the focusing of activations for trees (and also vanishing in combination with higher α). We looked for the models that had the best discrimination capacity as manifested by the largest number of unique winners in representing different vertices.

Despite the above mentioned commonalities, the models were observed to differ in their sensitivity to (α, β) setting, which depended on the data set. As shown in Figure 4.2, SOMSD reveals a rather irregular dependance (for all three data sets) of WD on (α, β) parameter values. On the other hand, in MSOM the "optimal" area roughly occupies a V-shape and differs in its α/β ratio for the three data sets. RecSOM shows a somewhat different dependance which again is quite systematic. Based on these graphs we selected optimal α and β for subsequent experiments. Figure 4.2 also demonstrates that it makes sense to search for optimal α and β independently, since these were found "off-diagonally", at least in case of RecSOM and SOMSD (for MSOM the optimal values lie close to the diagonal $\alpha + \beta = 1$).

Regarding (α, β) it could be argued that it would be more meaningful to change these parameters dynamically, rather than preset them to constant values. In particular, one could start with $\beta = 0$ to first force learning the leaf labels (not being disturbed by noisy contextual information), and later

[5] We were motivated by the experience with RecSOM in case of sequences where $\alpha > 1$ led to better representations (Tiňo et al., 2006).

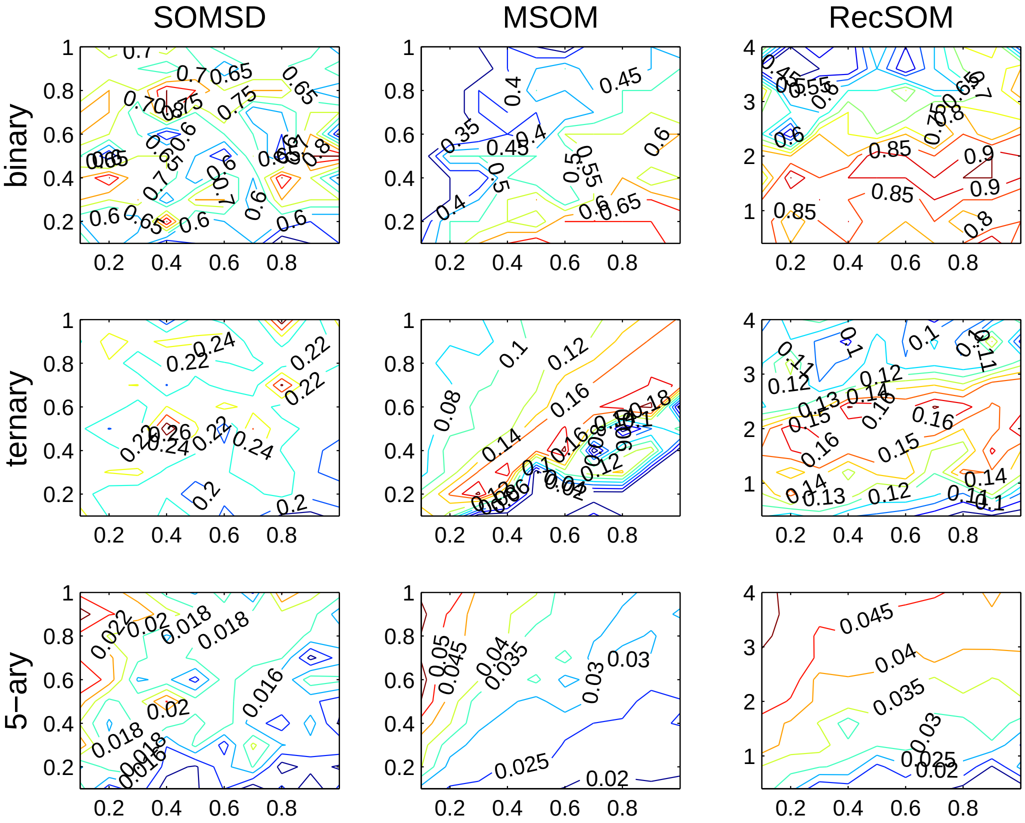

Figure 4.2: Mean WD as a function of systematically varied parameters α (vertical axis) and β (horizontal axis) for the three models trained on all three data sets. Whereas MSOM and RecSOM reveal systematic patterns in performance change (specific for each data set), SOMSD displays least evident order, with irregularly spaced small (α, β) islands with the highest WD.

increase β (while decreasing α) to also develop the model sensitivity to the structural information. [6] Although this argument is logical, the downside of it is that it prevents clear separation of leaves from trees in the map, because, as confirmed by our simulations, the winners for different leaves evenly spread over the map and then the winners for trees can only squeeze in between them (global reorganization does not take place). Hence, we preferred to simultaneously learn the labels and the structure by setting both α and β to

[6] Actually, this kind of strategy (entropy-controlled adaptation) with $\beta = 1 - \alpha$ was used in MSOM for processing the sequential data (Strickert and Hammer, 2005).

positive values, and to find the optimal values experimentally.

Once the optimal (α, β) parameters were found, 10 runs of the same model with random weight initialization from the interval [0, 1] were conducted and the mean values of measures were computed. In the case of the binary data set 100 runs were conducted.

We also tested larger maps with 225 (for binary data) and 400 (for ternary and 5-ary data) using the optimal parameters found for the initial map size. In MSOM, the context descriptor parameter was set to its default value $\gamma = 0.5$ in all experiments.

4.2.1 Binary syntactic trees

This data set was composed of 7 syntactic trees with labeled leaves and unlabeled inner vertices (vertices), having maximum tree depth 5 (the left half of Table 4.2). The trees were generated by a simple grammar originally developed for testing the representational capacity of the Recursive Auto-Associative Memory (Pollack, 1990). For the RAAM, being a two-layer perceptron trained as an auto-associator, the ability to represent a tree involves its successful encoding (at the hidden layer), and the subsequent unambiguous decoding (at the output layer). In case of our unsupervised feedback maps, on the contrary, there is only the encoding part. The ability of the map to represent a tree implies its ability to also uniquely represent all vertices (subtrees) contained in the training set (the right half of Table 4.2).

Table 4.2: Binary trees used for training and the list of non-trivial vertices comprised by the data set.

Training set		Vertices contained in it	
(d(a(a(an))))	((dn)v)	(an)	(p(dn))
((dn)(p(dn)))	((dn)(v(d(an))))	(a(an))	(d(an))
(v(dn))	((d(an))(v(p(dn))))	(a(a(an)))	(v(d(an)))
(p(d(an)))		(dn)	(v(p(dn)))

Similarly to RAAM, processing a tree in a feedback map proceeds bottom-up from the leaves, for which context activations are set to zero vectors, up to the root. When processing the inner vertices, the inputs are set to zeros. Intermediate results (activations) are stored in a buffer to be retrieved later. The weights are updated in each discrete step. The example is shown in Table 4.3.

The data set was presented during 2000 training epochs. During the first 1200 epochs, the neighborhood width was set to linearly decrease, σ:3→0.5

Table 4.3: The order of training inputs (organized in columns) in processing the tree ((dn)(p(dn))). The inputs (left context, label, right context) are mapped to output representations R (or R').

Input		Output	Input		Output
0 n 0	→	R[n]	0 d 0	→	R'[d]
0 d 0	→	R[d]	R[p] 0 R[(dn)]	→	R[p(dn)]
R[d] 0 R[n]	→	R[(dn)]	R'[d] 0 R'[n]	→	R'[(dn)]
0 p 0	→	R[p]	R'[(dn)] 0 R[(p(dn))]	→	R[tree]
0 n 0	→	R'[n]			

(ordering phase), and then kept constant over the next 800 epochs (fine-tuning phase). For the larger maps the initial neighborhood width was proportionally increased and the profile was kept the same. The leaves were assigned (symbolic) one-hot codes yielding the input space with dimensionality $M = 5$. For the best models of all sizes, we present the six quantitative measures (as defined in Section 4.1) in Tables 4.4–4.5 [7] and also the (typical) graphical information about unit weights and output activations. Standard deviations around the means were negligible in most cases, except for MED, QE and LWC measures in case of SOMSD.

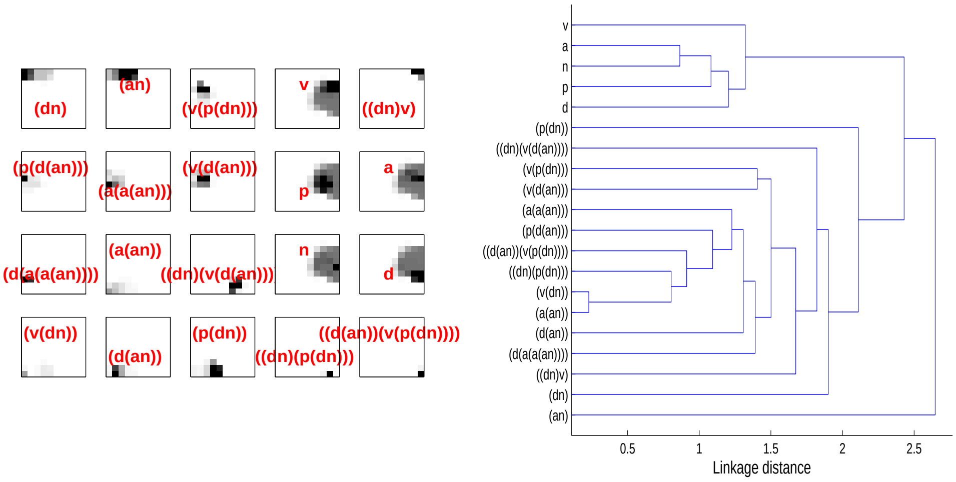

Figure 4.3: (a) Output activities of SOMSD and (b) the corresponding dendrogram for all vertices from the binary trees data set.

[7] In cases where standard deviations around the mean are not shown, they were negligible.

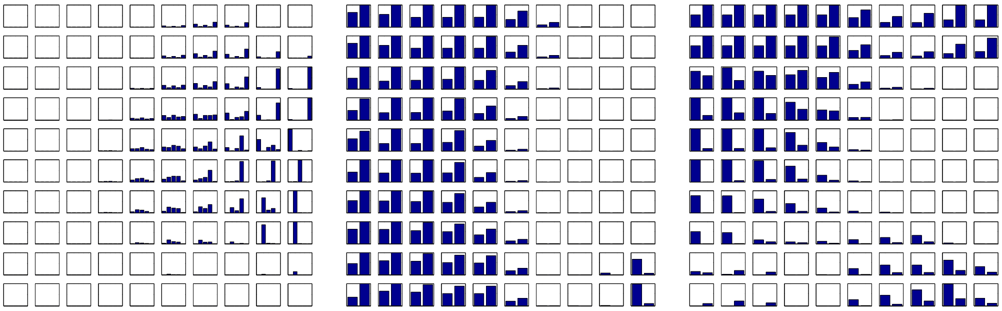

Figure 4.4: Converged (a) input, (b) left context and (c) right context weights of the SOMSD model trained on the binary trees data set. Topographic organization is evident in all cases.

SOMSD. The best results for SOMSD were obtained using $\alpha = 0.5$ and $\beta = 0.9$. The average fraction of unique winners was almost 0.93 (Table 4.5). The errors were caused by very similar trees (such as (a(an)) and (v(dn))) that share the winners. The SOMSD output activity in Figure 4.3a clearly shows the topographic organization. The winners for leaf vertices, located in the right middle part of the map, are clearly separated from winners for trees, occupying the complementary map regions. In addition, simpler trees, located in the upper left area, are separated from the more complex trees in the bottom right area of the map. It can also be observed that activation profiles for trees are more focused than those for leaves. The corresponding dendrogram of the map activity (Figure 4.3b) shows how the map differentiates between the trees. Leaves are clearly differentiated from non-trivial trees. [8]

The weight profiles (Figure 4.4) illustrate topographic ordering of all sets of weights. Their organization accounts for a clear division of map units in their specialization of representing either leaves (right middle area) or trees (the complementary parts of the map). Increasing the map size was observed to lead to an improvement (Tables 4.4 and 4.5): all 20 trees could be more-or-less reliably differentiated in the map.

MSOM. We chose $\alpha = 0.2$, $\beta = 1.0$. As seen in Figure 4.5a, winners for leaves (trivial trees) are again well separated from units representing trees. There are only WD = 0.80 of different winners, and each neuron becomes activated for several inputs. Hence, the MSOM activation is much more widespread than in the case of SOMSD (and RecSOM). Even though

[8] All dendrograms were created by the average linkage clustering method (applied to outputs $\mathbf{y}$), and the resulting hierarchical trees were optimized by a leaf ordering method that maximizes the sum of similarities of adjacent elements in the ordering (Bar-Joseph et al., 2001).

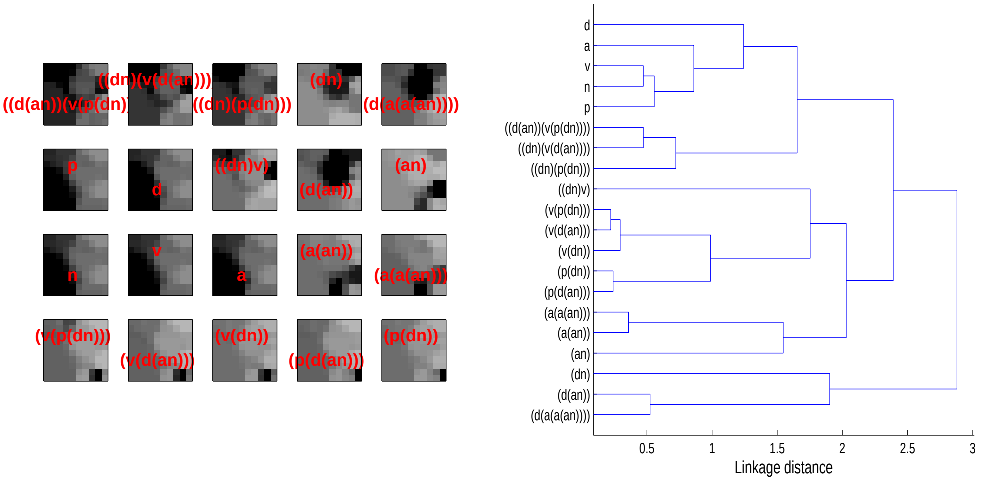

Figure 4.5: (a) Output activities of MSOM and (b) the corresponding dendrogram for all vertices from the binary trees data set.

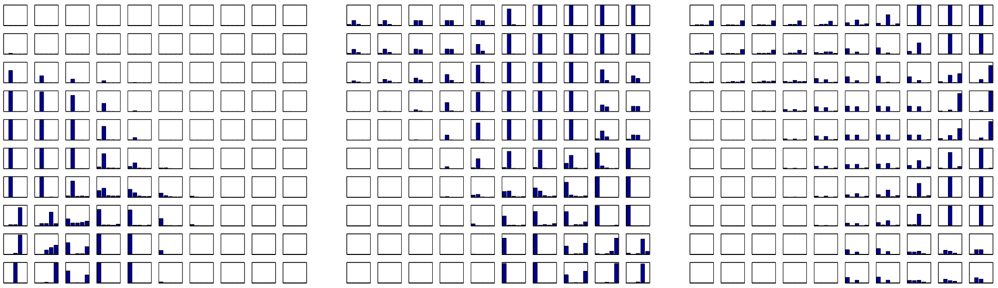

Figure 4.6: Converged (a) input, (b) left context and (c) right context weights of the MSOM model trained on the binary trees data set. Topographic organization is evident in all cases.

the dendrogram shows that MSOM differentiates between leaves and non-trivial trees, the trees are differentiated differently from SOMSD model. The weight profiles (Figure 4.6) show topographic order and splitting of the map in representing leaves and inner vertices. The differences between the left and the right contexts are due to asymmetry of the trees in the data set. Increasing the map size to 225 units led to an improvement of all measures (except MED that remained unchanged). The problem for all MSOMs (for every map size) was to differentiate between two trees: (v(d(an))) and (v(dn)).

RecSOM. Best results were achieved with $\alpha = 1.6$ and $\beta = 0.9$. This parameter combination differs from the other two models due to the opposite

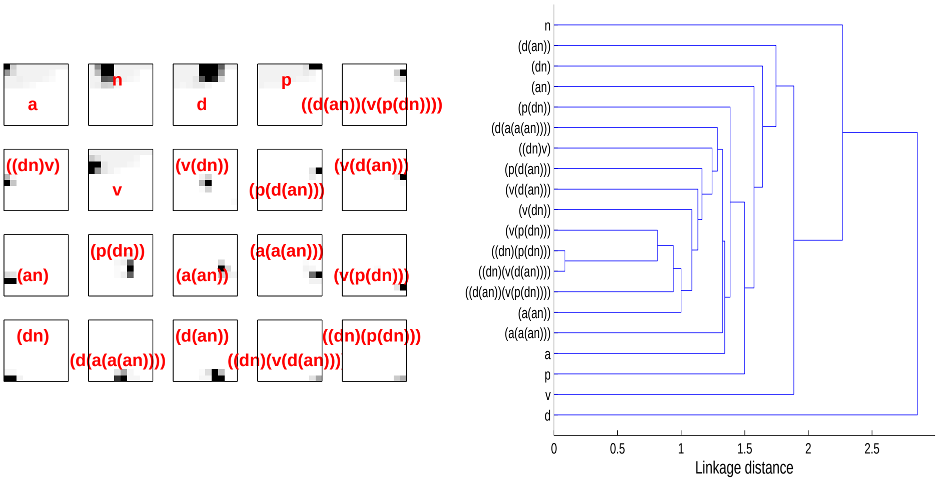

Figure 4.7: (a) Output activities of RecSOM and (b) the corresponding dendrogram for all vertices from the binary trees data set.

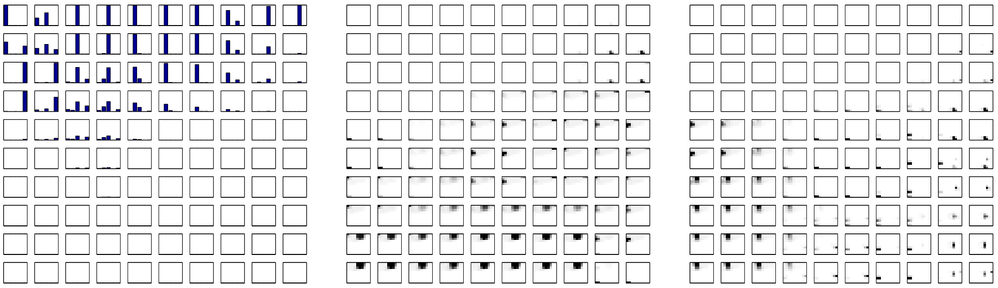

Figure 4.8: Converged (a) input, (b) left context and (c) right context weights of the RecSOM model trained on the binary trees data set. Topographic organization is evident in all cases. The context weights are displayed as 2D mesh plots.

α/β ratio. The likely reason is that higher value of α in RecSOM is needed to counterbalance the effect of high-dimensional context activations. Nevertheless, the output activations of leaf vertices remained very weak even after training. The whole map activity (Figure 4.7a) is more focused than in the case of MSOM and SOMSD but there are more different winners, 95% on average for the initial map size. The dendrogram of RecSOM (Figure 4.7b) is different from the previous models, the differentiation is not hierarchical and some leaves are mingled with trees. The reason lies in highly focused output activations. The weight profiles in Figure 4.8 show that units focus-

Table 4.4: Mean TQD and STQD measures for the models trained on the binary trees data set.

	10×10		15×15	
SOMSD	2.82	2.92	2.93	2.93
MSOM	2.61	2.66	2.86	2.86
RecSOM	2.83	2.86	2.93	2.93

Table 4.5: Mean WD and MED measures for the models trained on the binary trees data set.

	10×10		15×15	
SOMSD	0.93	0.003±0.002	1.00	0.0005
MSOM	0.80	0.002	0.95	0.002
RecSOM	0.95	0.0008	1.00	0.004

ing on leaves are again well separated from units focusing on trees. As with SOMSD, increasing the map size to 225 units led to maximum WD = 1.0 and to an increase of MED as well.

As seen in Table 4.6, QE also decreased in all larger maps, and MSOM showed the lowest QE. In terms of LWC, all models are equal, and the correlation between the winners and the locality of the trees was very high. LWC turned out to be insensitive to the map size.

4.2.2 Ternary linguistic propositions

The second set consists of ternary trees of linguistic propositions that were also tested in the case of linear RAAM model (Farkaš and Pokorný, 2007). This data set originated from English sentences that were generated using a specified probabilistic context-free grammar with semantic constraints and then rewritten into ternary propositions. The translation process resulted in 307 various (non-trivial) trees with maximum depth 7. Table 4.7 lists a few

Table 4.6: Mean QE and LWC measures for the models trained on the binary trees data set (LWCs for the larger maps is not shown since they remained unchanged).

	10×10		15×15
SOMSD	0.011±0.029	0.938±0.08	0.00055±0.00009
MSOM	0.00059	0.937	0.00031
RecSOM	0.0034	0.937	0.0041

examples of used sentences and their translations to propositions. [9] Some trees have empty inner vertices (NULL labels). The fifty words (i.e. leaves) in the lexicon implied $M = 50$ since the leaves were assigned localist (one-hot) encoding. We used the map with 15×15 units, and trained it for 20 epochs. The neighborhood size was set to σ:5→0.5 over 12 epochs and then remained constant. For the larger map, σ was proportionally increased. The learning rate was set μ:0.3→0.15 over 12 epochs, and further decreased linearly down to 0.1.

Table 4.7: Examples of simpler generated sentences and their translations.

Sentence	Proposition
Steve walks	(walks Steve NULL)
women see boys	(see women boys)
dogs who_pl see girl bark	(bark (are dogs (see dogs girl)) NULL)
boy feeds cat who John sees	(feeds boy (is cat (sees John cat)))

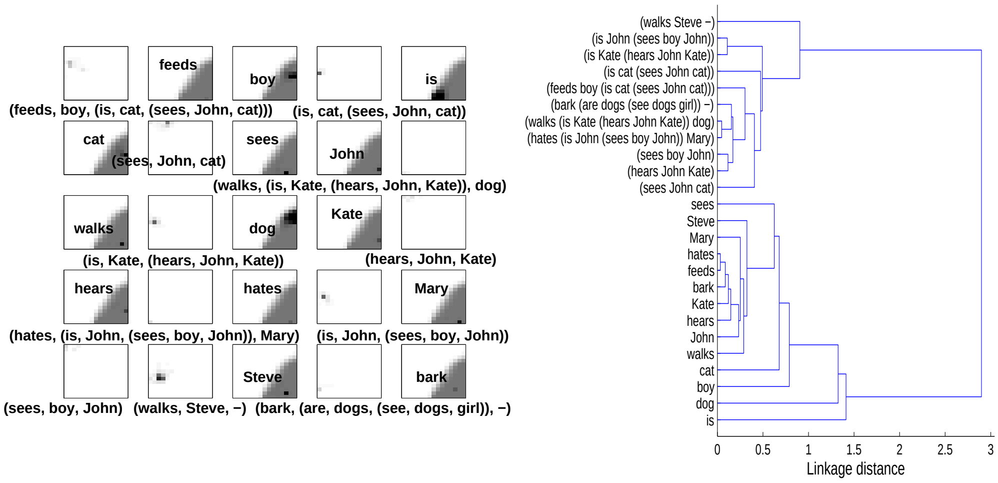

Figure 4.9: (a) Output activities of SOMSD and (b) the corresponding dendrogram for the 25 randomly selected vertices from the ternary trees data set. Longer tree labels in the activity map are positioned below the corresponding image.

SOMSD. We chose $\alpha = 0.5$ and $\beta = 0.4$. SOMSD clearly differentiates between leaves and non-trivial trees (Figure 4.9) but also clustering by tree

[9] In translation, the word who was replaced by the heads is and are depending on the number used in the phrase.

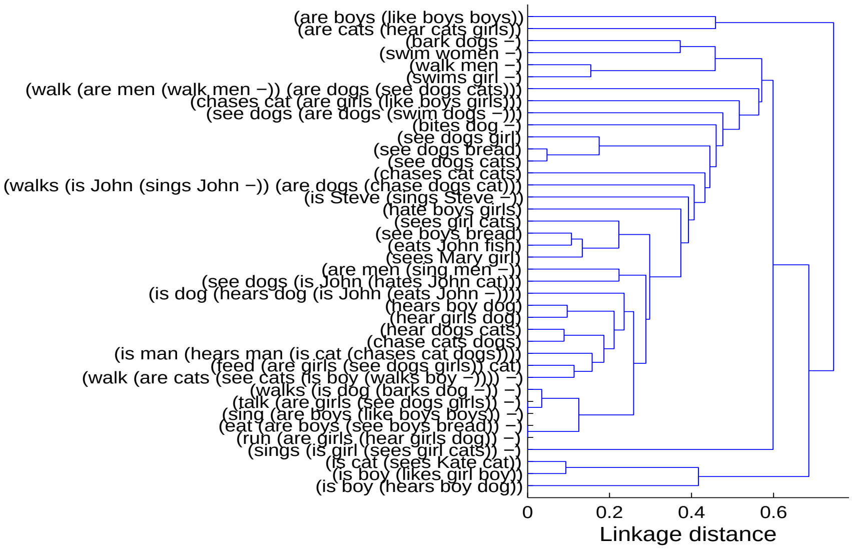

Figure 4.10: Dendrogram of the map activity for the 40 randomly selected non-trivial ternary trees. SOMSD differentiates trees based on the length and common RF.

complexity is visible. The winners for leaves are located in the lower right corner with little differences between their representations. In the case of leaves the activity of the map is less focused than in the case of more complex trees. The more complex a tree, the more focused activity is devoted to it in the map. SOMSD learned to differentiate non-trivial trees based on depth (Figure 4.10) and the topmost part of given trees, as stated in (Hammer et al., 2004b). For instance, a unit that happened to be the winner two input trees, (see dogs (is dog (bites dog NULL))) and (see cats (is boy (walks boy NULL))) has RF of the form (see – (is – –)) and SRF of the form (see dogs (is dog (bites dog –))). The length of RF reflects this feature in the other models as well, and is even more evident in case of polgen data set.

The number of winners in SOMSD was oscillating between 71 and 66 (in 30% of cases, there were 66 winners) and interestingly, the runs with fewer winners (corresponding to WD = 0.195) yielded better results in terms of TQD and STQD. Increasing the map size did not lead to MED > 0 which means that SOMSD cannot distinguish between some inputs even at the level of the map activity.

MSOM. The best results were achieved for $\alpha = 0.4$ and $\beta = 0.6$. Parameter β needed to be slightly larger than in SOMSD, because the context weights have higher dimension. It can be seen that MSOM activity (Figure 4.11a) is quite different from that of SOMSD. The whole map is activated for every input although most of the neurons show only low activity. Leaves are located in the lower right part of the map and non-trivial trees are scattered throughout the rest of the map. The dendrogram of the activations (Figure 4.11b) shows clear differentiation between trees and leaves. The trees with different depths are further differentiated.

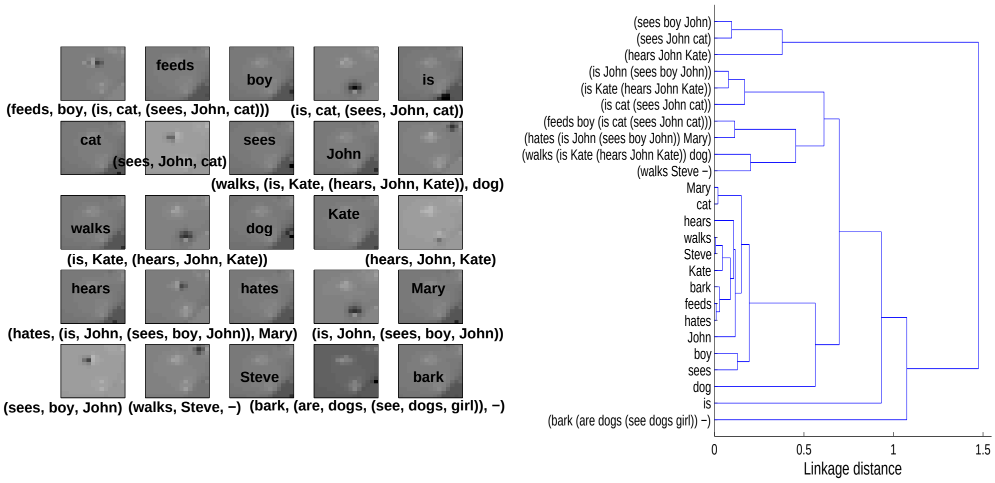

Figure 4.11: (a) Output activities of MSOM and (b) the corresponding dendrogram for the 25 randomly selected vertices from the ternary trees data set. Longer tree labels in the activity map are positioned below the corresponding image.

The dendrogram (Figure 4.12) with the same subset of 40 non-trivial trees (as in case of SOMSD) reveals that MSOM differentiates trees with respect to both input and depth. Regarding input, the first (topmost) word (being a verb) is the clustering parameter. The best result for MSOM was 53 distinct winners but the same problem as for SOMSD arises. Multiple runs with the same parameters led to two different results (53 and 24 winners). Increasing the map size led to decrease of WD (from 0.083 to 0.05), and only small changes in TQD and STQD. Based on these observations, we can claim that the performance of MSOM was the worst among all models for the ternary trees data set (also in terms of QE, as seen in Table 4.10).

RecSOM. The best results for RecSOM were achieved for $\alpha = 2.4$ and $\beta = 0.4$. Although the map activations (Figure 4.13a) looks similar to the previous models, the organization of output space is very different. Leaves are not located in one part of the map but are scattered across the map. The dendrogram (Figure 4.13b) uncovers the difference in organization. The

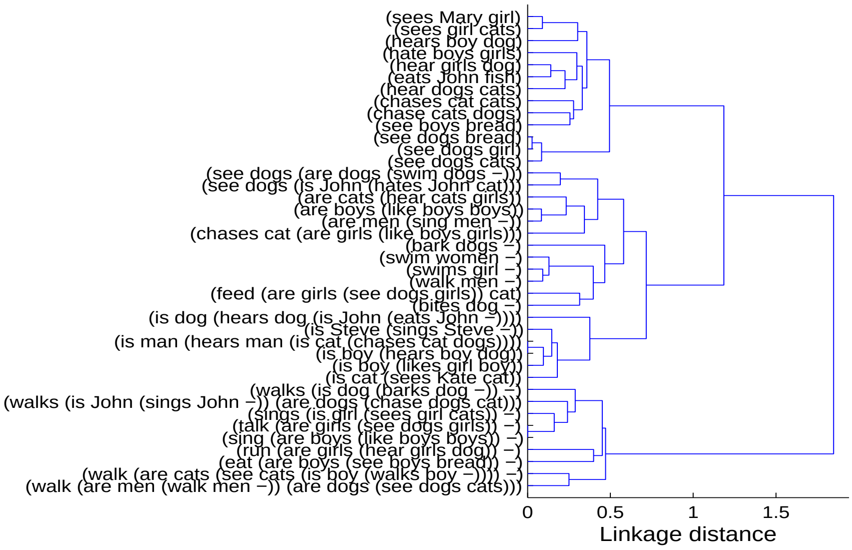

Figure 4.12: Dendrogram of the MSOM activity for the 40 randomly selected non-trivial trees. The map differentiates trees based on the length and the most recent input.

Table 4.8: Mean TQD and STQD measures for all models trained on the ternary trees data set.

	15×15		20×20	
SOMSD	1.10	2.65	1.14	2.90
MSOM	0.76	1.63	0.54	1.67
RecSOM	1.17	1.69	1.40	2.26

leaves usually have different activations than the rest of inputs. As in previous models, both the most recent inputs and the structure are important for the map activation.

The structure and RFs emphasis on resulting map activations of the 40 trees are shown in Figure 4.14. Both the structure and RFs are clearly visible. RecSOM typically had 57 winners (corresponding to WD = 0.16) and showed only minor differences between runs. In addition, RecSOM was the only model that achieved MED > 0, i.e. was able to differentiate every single vertex in terms of output map activation.

Regarding LWC (Table 4.10), the correlation between the winners and the trees is very low (and similar for all models) which might be due to higher depth of the ternary trees (and hence their higher potential variability).

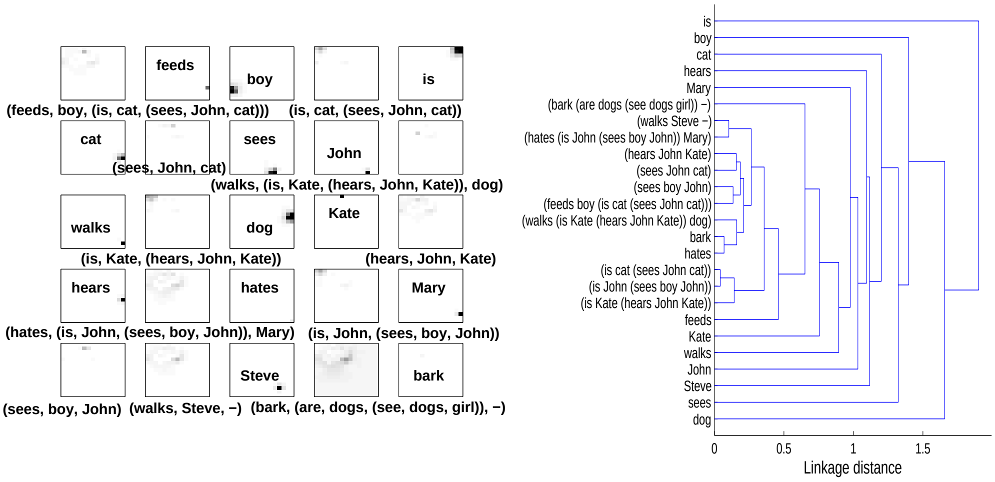

Figure 4.13: (a) Output activities of RecSOM and (b) the corresponding dendrogram for the first 25 vertices from the ternary trees data set. Longer tree labels in the activity map are positioned below the corresponding image.

Table 4.9: Mean WD and MED measures for all models trained on the ternary trees data set.

	15×15		20×20	
SOMSD	0.195	0.0	0.241	0.0
MSOM	0.083	0.0	0.050	0.0
RecSOM	0.160	$\sim 10^{-6}$	0.171	$\sim 10^{-6}$

4.2.3 5-ary graphical data

The third data set was created by Policemen Generator (polgen) (Hagenbuchner et al., 2003) using the house production rules. Contrary to the previous cases, this graphical data set contains 5-ary trees with labels on every vertex.

Table 4.10: Mean QE and LWC measures for the models trained on the ternary trees data set (LWCs for the larger maps are not shown, since they remained unchanged).

	15×15		20×20
SOMSD	0.22±0.027	0.18±0.035	0.100±0.019
MSOM	0.30±0.1	0.15	0.480
RecSOM	0.022	0.19	0.046

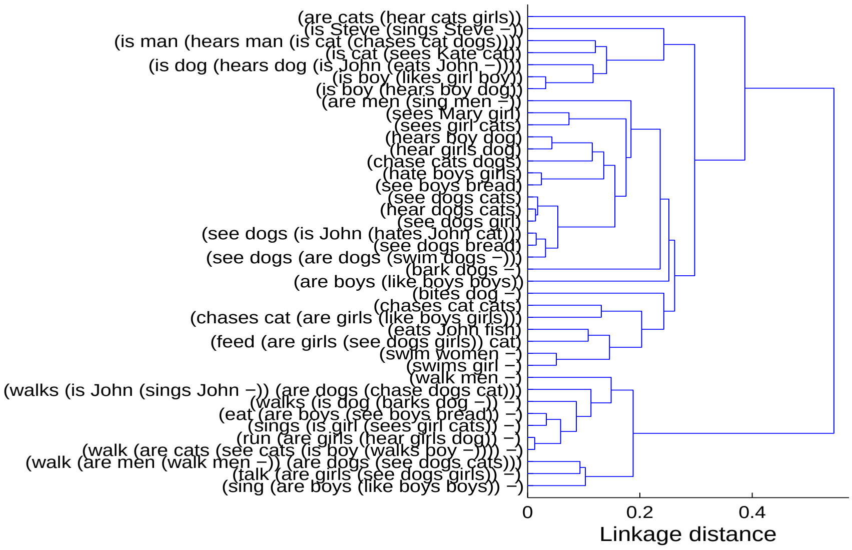

Figure 4.14: Dendrogram of the map activations for the 40 randomly selected non-trivial trees. RecSOM differentiates trees based on length and common receptive field.

For example:

(label1 (label2 (label7 9 9 9 9) label3 label4 label5 label6))

encodes the tree in Figure 4.15, where label1 is the root vertex with five children label2 through label6, and label2 has only one child, label7. [10].

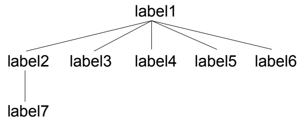

Figure 4.15: Example of a 5-ary tree used in the graphical data set.

Policemen Generator software package generates pictures based on the specified grammar. Basic shapes are used in the process of creating pictures. Pictures are further processed into graphs using included software. We generated 1000 pictures of houses using this grammar and converted them into trees. This data set contains 990 leaves [11] and 1493 non-trivial trees.

[10] Symbols '9' were inserted as padding to show that label7 is the leftmost child of label2 and the other children are empty

[11] Input vectors were generated using patgen (pattern generator provided in polgen software).

Five contexts are needed for training. The input vector size is set according to the generated input vectors to two. The map size for the 5-ary tree data is 15×15 and training is set to last 20 epochs. We set σ:5 →0.5 for ordering phase and then kept it constant. The learning rate was set to linearly decrease μ:0.3 →0.15 (ordering) and then further down to 0.1 (fine-tuning).

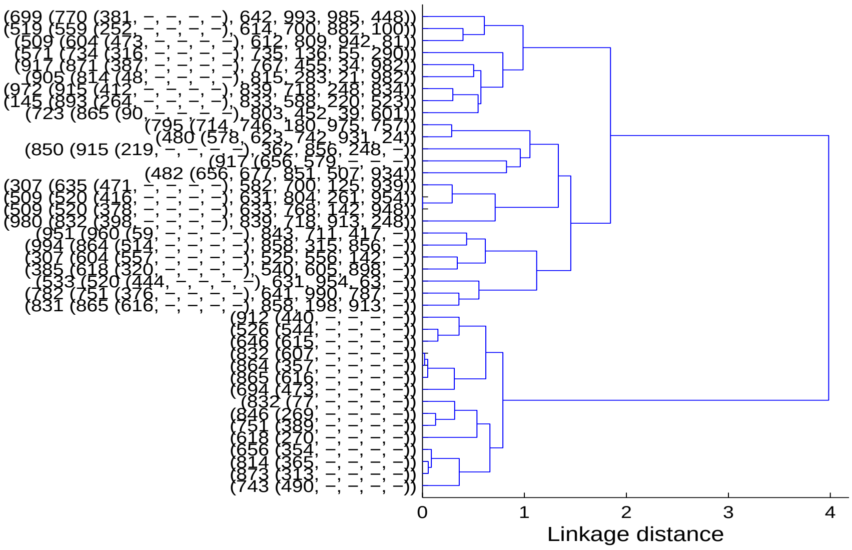

Figure 4.16: Dendrogram of SOMSD for the 40 randomly selected non-trivial 5-ary trees.

SOMSD. We set $\alpha = 0.9$ and $\beta = 0.1$ as optimal parameter values. The reason for $\alpha > \beta$ lies in the large amount of leaves in the data set (almost 40%). The percentage of vertices SOMSD is able to successfully represent is rather low (polgen data is very large). More than half of the neurons do not become winners for any input. Figure 4.16 shows the differentiation between the trees more clearly. Leaves were removed from the dendrogram for better comparison between the tree structures. The dendrogram is divided into two parts – the first part consists of the trees containing only one child and the second part consists of more complex trees. The more complex trees are differentiated less between each other, but some differences can be seen. The trees with similar structure and similar root value are closer to one another. Apparently, the structure is more important than the root values, but root values also play a role when the structure is the same. Even though TQD = 0.0072 is very low, STQD = 3.17 is higher than in the cases of binary and ternary trees. The reason is the large amount of leaves in this data set which affects TQD but not STQD. For the larger map (20×20) the results are a little better, and WD is very low even for the smaller map, and is roughly doubled for the larger map. MED = 0 for both map sizes, because there are different input patterns with the same activations of the map.

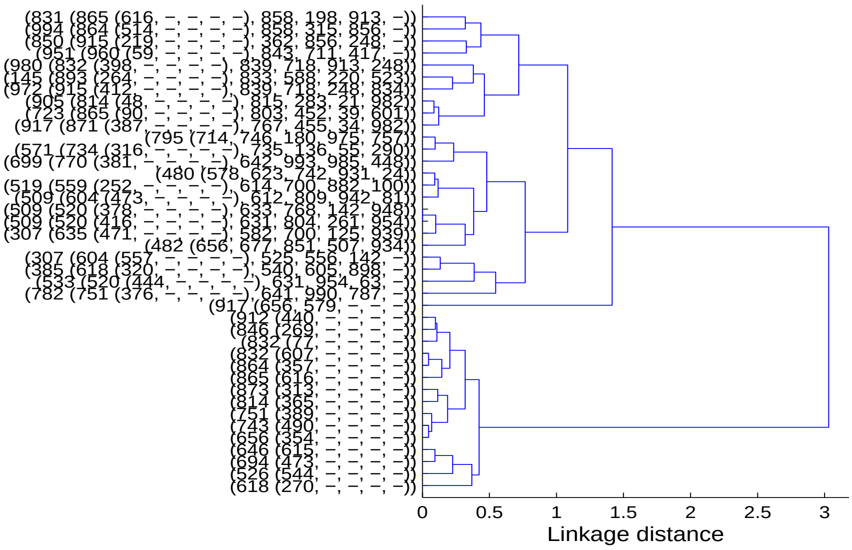

Figure 4.17: Dendrogram of MSOM for the 40 randomly selected non-trivial 5-ary trees.

MSOM. For optimal values $\alpha = 0.6$ and $\beta = 0.1$, MSOM yielded worse results than SOMSD regarding memory depth. WD is slightly higher than in other models. On average MSOM had 153 winners for 15×15 map. The dendrogram (Figure 4.17) reveals how MSOM encodes the structures. There are very small merging distances (less than 0.5) within the class of the simplest trees with only one child. The structure is more pronounced than the root element in this case and it is visible for clusters of the same structure but different labels. For the 20×20 map the measure TQD increased (in other models this measure decreased) to a high value 0.066 and STQD only slightly increased. This is caused by more winners in the larger map, in this case 239, which corresponds to WD = 0.096. This value is highest among all models for this map size and corresponds to 59.7% winners in the map. In the case of MSOM, again MED = 0 (for both map sizes).

RecSOM. This model performs best on this data set with $\alpha = 3.6$ and $\beta = 0.1$, for which TQD = 0.0002, being the smallest value among the models. Only for this model, MED > 0. WD = 0.053 corresponds to 132 winners (out of 225). The map output activation (Figure 4.18) of randomly selected

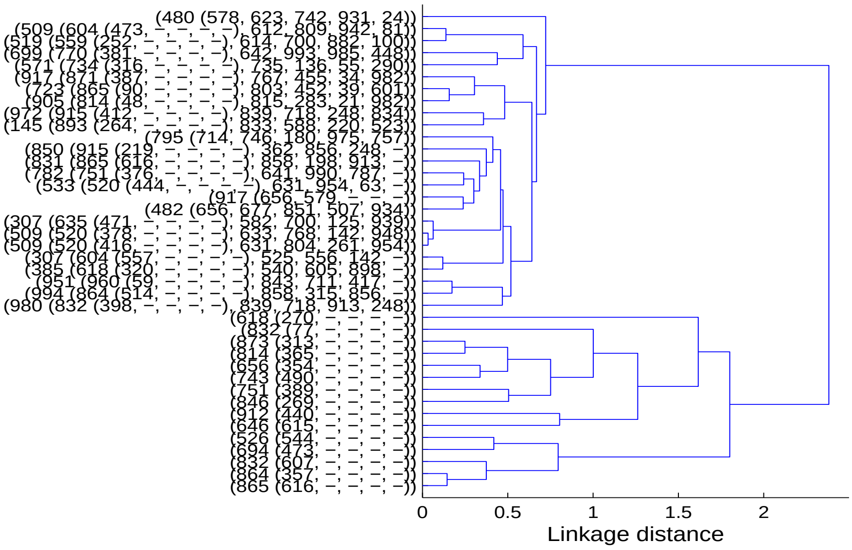

Figure 4.18: Dendrogram of RecSOM for the 40 randomly selected non-trivial 5-ary trees.

trees (leaves were omitted) reflects the differences compared to previous models. Even though simple trees with one child are strictly differentiated, there are differences between them which means different winners. Also more complex trees are differentiated into clusters but not all clusters share the same structure properties. Some clusters contain mixed tree structures and the mixed root vertex values. Hence, the clustering properties of RecSOM on this data set are not completely clear. The enlarged map provides slightly better results regarding most measures as seen in Tables 4.11–4.13 (TQD, STQD and WD increased, while QE decreased). For both map sizes, MED > 0 which implies that every vertex is distinguished by the map. Hence, RecSOM is the only model that can differentiate every input on the data set that is 11 times larger than the number of neurons in the map.

Table 4.11: Mean TQD and STQD measures for all models trained on the 5-ary graphical data set.

	15×15		20×20	
SOMSD	0.0072	3.17	0.003	3.18
MSOM	0.0013	2.49	0.066	2.52
RecSOM	0.0002	2.88	0.001	2.90

Table 4.12: Mean WD and MED measures for all models trained on the 5-ary graphical data set.

	15×15		20×20	
SOMSD	0.044	0.0	0.071	0.0
MSOM	0.062	0.0	0.096	0.0
RecSOM	0.053	$\sim 10^{-7}$	0.071	$\sim 10^{-6}$

Table 4.13: Mean QE and LWC measures for the models trained on the 5-ary graphical data set (LWCs for the larger maps are not shown, since they remained unchanged).

	15×15		20×20
SOMSD	0.001	0.373	0.00074
MSOM	0.00027	0.371	0.00019
RecSOM	0.0017	0.371	0.00160

4.3 Summary

We can summarize all these results as follows: SOMSD, unlike MSOM and RecSOM, is by far more sensitive (i.e. less robust) to weight initialization. Regarding the memory depth, SOMSD and RecSOM clearly outperform MSOM but only in case of binary and ternary trees. SOMSD turns out to best cluster structural information. RecSOM clearly outperforms the other models in terms of inputs discrimination, based on map activation. MSOM consistently yield the lowest QE and all models are equal in terms of LWC.

The best and the worst data sets for the compared models:

Model	Ideal data set	Worst data set
SOMSD	simple data set	complex data set (big fan-out)
MSOM	more complex data set	simple and small data set
RecSOM	complex and large data set	simple and small data set

Table 4.14: Summary of best and worst data sets for each model based on experiments.

The difference between worst data sets for MSOM and RecSOM is that while the worst data set for RecSOM is caused by the complexity of this model, the worst data set for MSOM is caused by its results.

The ideal job for these models are as follows: SOMSD – clustering and structure processing, MSOM – clustering of complex data and RecSOM – memory simulation and data recovery.

Chapter 5

Processing structured data from XML

Visualization of tree structures is a suitable task for recursive SOMs. We illustrate the viability of this approach using a concrete example of visualization of tree structures contained in simple XML files and its practical usage in information extraction for large XML files.

Some experiments (Hagenbuchner et al., 2005; Kc et al., 2006) have already been conducted on XML data set model using SOMSD model. In that case the data set consisted of very complex XML files that needed to be preprocessed and scaled down to be used for training. The trained map was used to cluster and visualize trained map for SOMSD model.

5.1 XML format

Data sets can be encoded in various formats that allow structural and item information to be saved. Although special proprietary formats can be used there are also standard formats. Native format for tree structures that is complex enough to be used for different types of data sets is XML format. XML format can be machine read, can be machine processed and validated. However, XML format is not able to encode cyclic graphs and for such data sets another format is needed. Also there is quite a lot of data already represented in XML format ready for processing. Another easy way of getting XML format data set is from database export.

The format of any XML file can be defined through Document Type Definition (DTD) file. Although DTD files support only rudimentary data types these are usually enough for describing most of the XML files. Using DTD the structure of the XML file can be seen. It defines, in graph language,

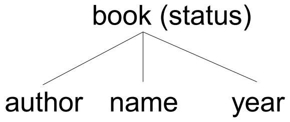

Figure 5.1: Tree representation of the DTD for the first data set. It is a complete ternary tree structure of depth one.

what is the structure of trees that can be created in the particular XML file. We use DTD to show the tree structure of the processed data.

We focus on two SOM models with feedback connections: SOMSD that uses reference only to the winner index and MSOM that refers to the winner content. We further focus on a simple XML data set as a proof of concept for data visualization and a complex XML data set for information extraction.

5.2 Experiments

The first data set used to illustrate viability of the models on XML files was arbitrary XML file consisting of library data about books. The structure of the XML file can be described by the following DTD:

```
<!ELEMENT library ( book+ ) >
<!ATTLIST book status NMTOKEN #REQUIRED >
<!ELEMENT book ( author, name, year ) >
<!ELEMENT author ( #PCDATA ) >
<!ELEMENT name ( #PCDATA ) >
<!ELEMENT year ( #PCDATA ) >
```

This DTD can be visualized as a forest, a group of trees. Fig. 5.1 shows the tree structure of one of the trees in the group.

An attribute of element book was processed as well as elements author, name and year. The set of trees created this way were simply ternary trees.

AN XML file consists of 6 records (trees) to show visualization that is available when using recursive SOM models. These 6 records contain 13 different elements (labels). The whole data set size is 24 with 19 unique labels. SOMSD model was chosen to be trained on this data set as this model retains more information about structures (Vančo and Farkaš, 2009). This is a desirable attribute for visualization of results. MSOM model can be used as well.

The size of the map was set to 5 × 5 for visualization purposes. Larger maps would differentiate worse than smaller maps. The inputs were presented

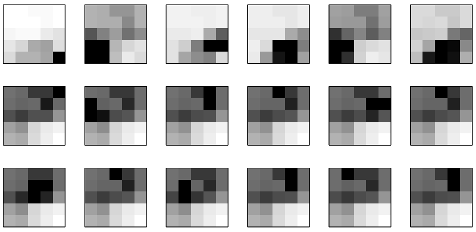

Figure 5.2: Activities evoked by different inputs in trained SOMSD map. The top row shows map activity as a response to the complete trees and the next two rows show activities of elements in XML.

for 2000 epochs. The neighborhood size was set to 3.0 decreasing to 0.5 during 1200 epochs and then kept stable at 0.5 for more stable organization. Learning rate started at 0.3 and decreasing over the same time to 0.15 and then decreasing to 0.1 over the last 800 epochs. The inputs were localistically encoded into vectors of size 15.

Parameters α and β were set experimentally searching $\langle 0, 1 \rangle$ space in 0.1 steps independently. The best results (the most active neurons) were achieved with parameters $\alpha = 0.5$ and $\beta = 0.1$.

There are multiple types of visualization of resulting trained map. We chose activity of the map for chosen input. It shows how the map organizes input after training. Close activities of different inputs suggest inputs that have commonalities and activities of different inputs that are far apart suggest inputs that are different. As map is trained using unsupervised learning only input information is used to categorize data.

In Figure 5.2 map output activity for the chosen inputs are presented. In the first row non-trivial trees are shown and these can be visually separated from elements in the last two rows. Activities of elements were located in the upper part and non-trivial trees had activities located in the lower part. This separation was expected. Separation of non-trivial trees was more important.

Non-trivial trees were separated in this data set according to the content of the tree (Fig. 5.3). On the left, trees with the same root (*sold*) are compared. These had very different activities – the highest activities were located in bottom corners. On the right, trees with different roots but the same children are compared. In this case, as expected, the highest activities were closer both located in the lower left part of the map.

Complex structured data trained without any prior knowledge on recursive maps lead to differentiating activities of input data into clusters that can be visually detected and separated. This visualization can be used for

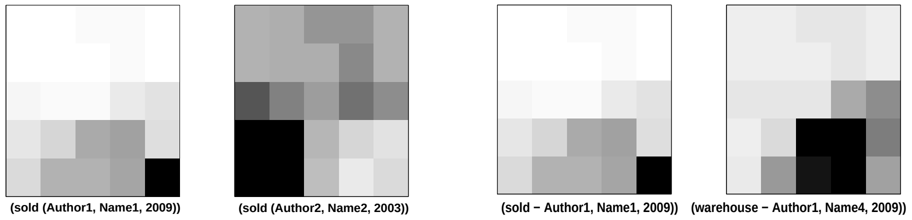

Figure 5.3: Differences in activities depending on input on a trained SOMSD map. Activities of two different inputs with the same root (left), two similar inputs with the different root (right). Visualization reveals the difference that can be hidden in the data set.

any data set extracted from XML file trained on recursive maps.

5.3 Large XML File

The large XML file was from XML Data Repository University of Washington [1]. This data repository consists of publicly available datasets in XML form. We decided to choose SIGMOD record data set as it was sufficiently large and was in a right format for easy processing. SIGMOD data set is an index of articles from ACM SIGMOD Record from 2001. Data in the XML file are information about issues and articles from ACM SIGMOD.

We chose only part of the XML consisting of information about articles. We ignored information about issues. The corresponding DTD for this part was defined:

```
<!ELEMENT articles (article)* >
<!ELEMENT article (title,initPage,endPage,authors) >
<!ELEMENT title (#PCDATA)>
<!ELEMENT initPage (#PCDATA)>
<!ELEMENT endPage (#PCDATA)>
<!ELEMENT authors (author)* >
<!ELEMENT author (#PCDATA)>
<!ATTLIST author position CDATA #IMPLIED>
```

In the SIGMOD XML file no attributes are defined other than a position of an author that was ignored but can be used as a supplementary information for more precise information extraction. The tree structure extracted from XML file was trimmed to maximum out-degree of 4 and therefore 4-ary trees were created.

[1] http://www.cs.washington.edu/research/projects/xmltk/xmldata/

There are no values in the inner vertices of the tree and empty vertices were used to encode these values. Information is only in the leaves of the tree. There are 1504 records in the SIGMOD XML file. The data set consists of 10832 trees and labels with 6624 unique values. Out of these are 2773 non-trivial trees. Model chosen for this data set was MSOM as this model differentiates inputs better than SOMSD model (Vančo and Farkaš, 2009).

The size of the map in case of larger XML file was 15 × 15 as the data set is comparably larger than in the previous case. Inputs were presented during 20 epochs. Neighborhood size was set to 5.0 decreasing to 0.5 during 12 epochs and then kept stable at 0.5 for more stable organization. Learning rate started at 0.3 and decreasing over the same time to 0.15 and then decreasing to 0.1 over the last 8 epochs. Inputs were encoded into vectors of size 2. The best results for MSOM model for the large XML data set yielded parameters $\alpha = 0.6$ and $\beta = 0.1$.

There are 164 active neurons (winners for at least one input) out of 225 neurons. These create 164 clusters of inputs that can be further processed and analyzed. Using the map size of 225 on 6624 unique inputs pushes the map to cluster similar inputs. Information about different clusters can be used to reduce the input dimension and extract similarities between inputs in the same cluster.

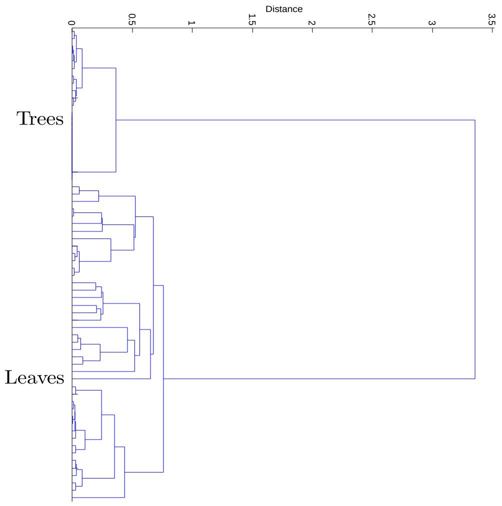

Figure 5.4: Dendrogram of inputs in the trained map. Two separate clusters can be seen: The cluster of trees (left) and the cluster of elements/leaves (right). The merging distance is a little less than 3.5.

To see the clustering performed by the trained MSOM, dendrograms for inputs were created. In the first dendrogram (Fig. 5.4) separation of leaves

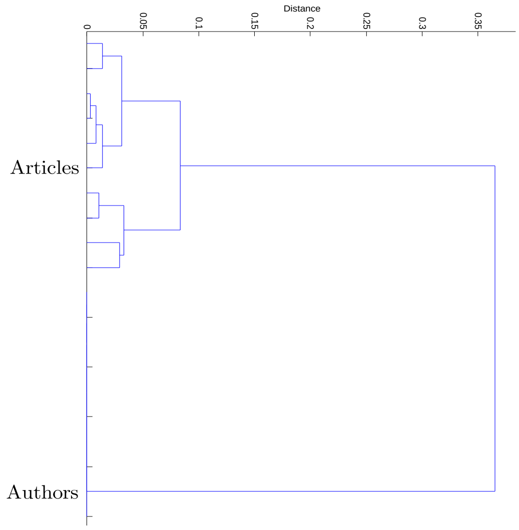

Figure 5.5: Dendrogram of tree inputs in the trained map. Only trees were selected and scaled in. Again two clusters can be seen: the cluster of articles (left) and the cluster of authors (the simplest trees) (right). The left cluster of articles can be further zoomed in and analyzed.

and trees takes place. Clusters of leaves and trees can be further analyzed. We decided to focus on trees. Dendrogram (Fig. 5.5) shows separation of simple trees consisting of author information and complex trees consisting of the whole article data. As seen the tress with author information were inseparable and formed one cluster.

Further inspection of the inputs is needed to extract the clustering information.

5.4 Summary

Recursive SOM models can be exploited for an easier information extraction and for a visualization of input data. Clustering of the XML data can lead to creating new views on the known data set.

Using the XML format for input data brings advantages of this format (syntax validation, parsing) to structured data it is representing. Combining XML format and recursive SOM models quick and easy unsupervised processing can be created.

However there are some problems encountered while using XML format. Attributes as well as element information has to be merged to create a unique name. Empty label has to be defined when no information is available.

Chapter 6

Batch learning

The idea of batch learning in contrast to classic, online, learning is to present input in a batch and not linearly, one input at a time. This approach is quicker than online learning as the learning process is run after the whole batch rather than after every input. In case of neural networks the weight update is a time consuming operation and therefore the time complexity of the batch learning is lower than that of online learning. The weights are updated after the whole data set has been presented. Therefore the update algorithm has to take all changes into account.

Another advantage of batch learning is a direct consequence of the batch type of processing. It is the usage of distributed computing – to use multiple computers on the computation of changes. The online learning has to be processed linearly but in batch learning the ordering is not important. That means that changes can be computed independently from each other for every input. In other words the data set can be split into parts by a master computer and provided to the slave computers along with the current state of the model. The slave computers then send their results (changes) for their inputs to the master computer which computes the new weights according to the provided changes, and the new epoch of the learning can start.

Not all neural network models have batch version of the learning defined. Where the batch learning is properly defined it is in the most cases the better learning algorithm due to its time efficiency and distributed computing advantage mentioned earlier.

The simplest neural network model using batch learning algorithm is a perceptron. Batch learning in this case is exemplary. Online learning takes input, computes output and changes weights according to difference between output and requested output. During batch learning all differences and corresponding inputs are saved and used for weights update at the end of the epoch.

For the simple recurrent network a simple batch learning exists as well. Multiple online learning methods exist for the simple recurrent network. Batch learning is defined only for back propagation through time (BPTT) (Haykin, 1999). The batch version of the BPTT algorithm is called epochwise BPTT. Its main steps are:

1. forward pass through the data set for every input, responses for the corresponding inputs are saved
2. backward pass to compute local gradients based on saved responses
3. weights adjustment based on computed local gradients

The same technique is used in batch learning of other models with the difference in the second step. The second step is specific for the model for which batch learning is defined. The precise algorithm with all equations can be found in Haykin (1999), chapter 15.

6.1 Batch SOM

Kohonen (2001) proposed batch learning version for the SOM. If we take one neuron from SOM and compute the influence regions of all results for this neuron and compute the mean over union of these regions we get a new state of the neuron in the map. This process can be done for all neurons. For every input the information that has to be saved is: the amount of influence for every neuron, the winner and the input. Batch learning can be described as

$$\mathbf{w}_i = \frac{\sum_{t=1}^{m} h(i, i^*)(t)\mathbf{x}(t)}{\sum_{t=1}^{m} h(i, i^*)(t)} \tag{6.1}$$

where m is the data set size, $\mathbf{x}(t)$ is the input at time t, $h(i, i^*)(t)$ is the neighborhood function between i-th neuron and the winner at time t.

The learning rate parameter is not used in batch learning algorithm [1] and therefore there is no need to optimize this parameter. The only parameter influencing the learning is the neighborhood size (and its change through time). Another important difference between learning algorithms is that weights are updated without taking old values into account in the case of batch learning. Due to this property convergence is faster in batch learning than in online learning. That means less epochs are needed for batch learning

[1] Learning rate parameter is removed in the process of creating Eq. 6.1

to achieve similar results to online learning. In online learning the change of weight is affected by the old value of the weight. The correctness of batch learning algorithm (convergence and ordering of weights) was proved (Cheng, 1997). Batch learning optimizes the same cost function as its online variant.

Both these learning algorithms are describing the same process of weight ordering (Kohonen, 2001). Despite that the resulting trained map using online and batch learning can differ.

In case of batch learning the space complexity is asymptotically the same as for online learning (Eq. 1.9) as only the map weigths and the updates of all weights are stored in memory at any given time:

$$S(\mathrm{SOM}^{batch}) = O(Nn + Nn + c) = O(Nn) = S(\mathrm{SOM}) \tag{6.2}$$

More important, the time complexity for one input, shows asymptotically the same results as Eq. 1.10 even though the weight update step was removed:

$$T(\mathrm{SOM}^{batch}_{input}) = O(n + Nn^2 + N + c) = O(Nn^2) \tag{6.3}$$

The weight update step runs after every epoch, therefore the time complexity of the batch SOM is lower but only by a constant:

$$T(\mathrm{SOM}^{batch}) = O(pT(\mathrm{SOM}^{batch}_{epoch})) = O(p(mT(\mathrm{SOM}^{batch}_{input}) + Nn^2))$$

$$T(\mathrm{SOM}^{batch}) = O(pmNn^2 + pNn^2) = O(pmNn^2) \tag{6.4}$$

The batch SOM has all the advantages of batch learning: faster and distributed computation. The difference in speed over epoch is less pronounced in this model as the complexity of the computation lies more within computing the winner than updating the weights. The important property is faster convergence (only few epochs suffice for the model to converge) and distributed computing. The distributed computing can be used to compute very large map sizes in shorter time on multiple computers.

But there are also some disadvantages to batch SOM. As mentioned earlier the disadvantages are connected with difference between batch and the online version. For batch SOM its learning algorithm is the advantage as well as disadvantage. On one hand the algorithm makes computation faster and fewer epochs are needed for the convergence. On the other side batch learning has different results than online SOM, the data is not well organized, the resulting trained map is not correctly ordered (Fort et al., 2002). There is one more disadvantage and that can be demonstrated using localist encoding.

6.2 Localist encoding and batch SOM

Batch learning fails if the data set is symbolic and has one hot (localist) encoding of inputs (Vančo, 2009a). As an example data set we used Tic Tac Toe data set (University of California Irvine, 2009). The results of using batch SOM on such encoded input data are considerably worse than using the online version of SOM algorithm.

This data set contains combination of states of the game Tic Tac Toe (Fig. 6.1) as well as information about winning or losing of the starting player. Data set contains 958 games of which 626 are successful games (win) and the other games are unsuccessful (lose) for the starting player.

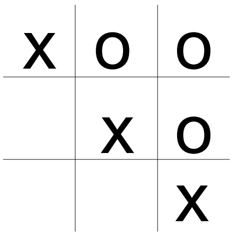

Figure 6.1: Final state of the game Tic Tac Toe. The first player (symbol X) won this game as he got three his symbols in one diagonal.

Although this information is not used during learning, it is used in comparison of the results. This data set is complex from various standpoints. Win or lose state occurs when the same mark is placed in a row, in a column or either of the two diagonals.

In the encoding that was used the game plan is linearized and the information about column and diagonal wins is partially lost. In the linearized version column and diagonal wins are not clearly visible. For example the game plan in Fig. 6.1 is linearized to the following vector:

x o o b x o b b x

where b means blank space.

For the encoding into the neural network localist code of length 3 for every symbol was used. The length of input vector was 27.

For comparison of both learning algorithms we used 5 different map sizes (10×10, 15×15, 20×20, 25×25 and 30×30). As batch learning has quicker convergence every map was trained using multiple number of epochs (100, 200, 300, 400 and 500 epochs). The learning rate parameter (for online learning) is the same for all models. Starting with value 0.5 linearly decreases to value 0.2 during the first 70% of epochs (ordering phase) and then linearly decreased to value 0.1 to the end (fine-tuning phase).

Parameter σ (neighborhood size) linearly decreased during the first 70% of epochs to 1.5 from the starting value depending on the map size. For the map size 10×10 the starting value was 3.0, for the map size 15×15 it was 5.0, for the map size 20×20 it was 7.0, for the map size 25×25 it was 9.0 and for the map size 30×30 it was 11.0 Training of every map (25 different maps) was repeated 100 times with random initialized map weights using interval $(0, 1)$. Results were averaged and compared using chosen measures.

6.2.1 Measures

We used two numeric measures for comparing results of the experiments.

- WD – level of winners computed as the ratio of the number of different winners for the entire data set and the size of the data set (Eq. 4.3 defined in section 4.1).
- $\mathrm{WD}_{\mathrm{map}}$ – level of winners computed as the ratio of the number of different winners for the entire data set and the map size

$$\mathrm{WD}_{\mathrm{map}} = \frac{|\{j \mid \exists t : j = i^*(t)\}|}{|N|} \tag{6.5}$$

Trained maps were visually inspected to see the distribution of the active neurons. Also winner distribution was compared according to the won or lost status of the input. Active neurons were labeled and the resulted maps were analyzed.

6.2.2 Results

Online learning

Optimal result (the highest number of active neurons) for the map size 10×10 was achieved over 400 epochs. Longer learning (500 epochs) had worse result. Analysis of the trained map (Fig. 6.2) shows equal distribution of the activity for the first component, the second and the third component are equally distributed as well. For all three components are visible multiple centers of lower values (white color). The first three components represent the first symbol in the game (the top left corner on the game board).

Increasing map size to 15×15 increased number of active neurons to 54, that is 24 % of all neurons. The best result was achieved with training over

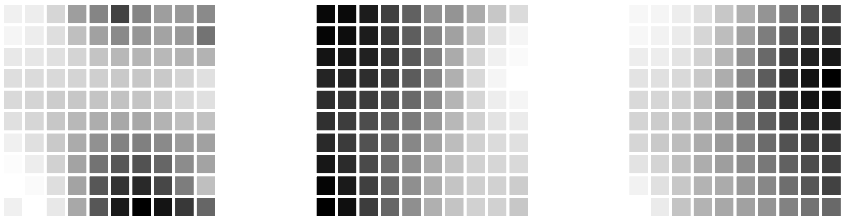

Figure 6.2: The first three components of weight vectors of the trained map 10×10 over 400 epochs using online learning. For every neuron its first (left), second (middle) and third (right) components of weight vector are shown. Darker squares mean value is closer to 1, lighter squares mean value is closer to 0.

300 epochs. Increasing size of the map did not bring increase in the per cent of active neurons for this data set. Map size of 20×20 had 102 active neurons over 200 epochs. In this case number of active neurons is twice as high as in the case of map 15×15. For map size 25×25 training over 200 epochs had the highest number of 104 active neurons. This increase is very low again.

The best result (in the case of the number of active neurons) yielded map size 30×30 over 400 epochs, 119. The first three components of the trained map of this size are shown (Fig. 6.3) for the visual analysis.

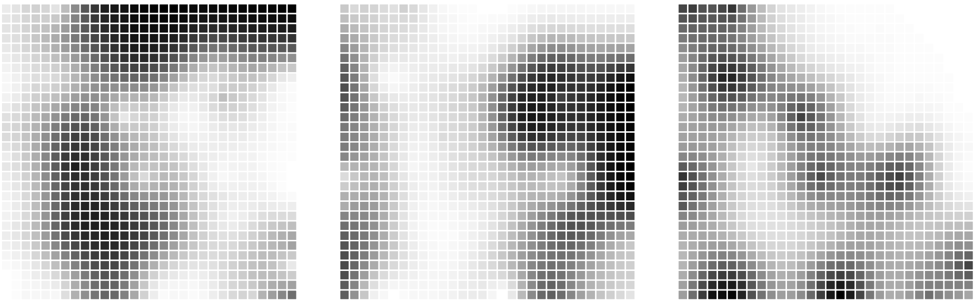

Figure 6.3: The first three components of weight vectors of the trained map 30×30 over 400 epochs using online learning. For every neuron its first (left), second (middle) and third (right) components of weight vector are shown. Darker squares mean value is closer to 1, lighter squares mean value is closer to 0.

The higher map size shows the activity of the trained map better than in the case of a small map size. The fitting of the components is better seen, that means that the sum of the first three components (and the next three components, etc.) equals almost the same value (around 1). This effect is caused by input encoding where only one value of these three components is 1, the other values are 0 (localist encoding). The activity is not equally distributed and areas with higher value are located in different parts of the map.

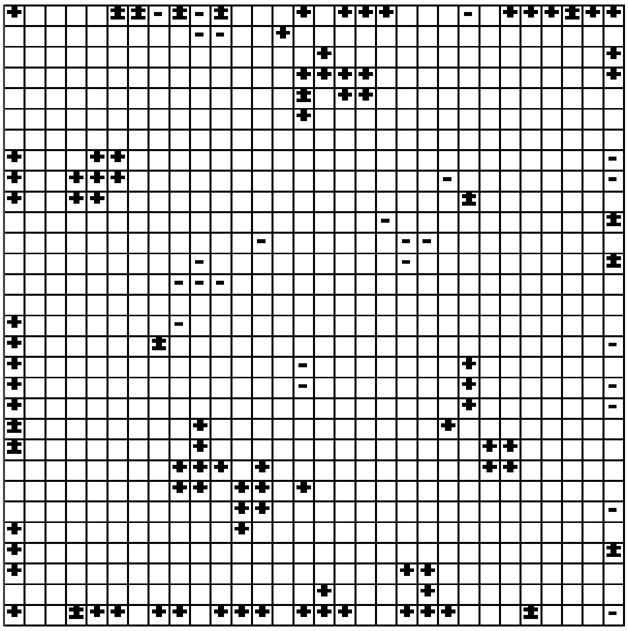

Figure 6.4: Graphical representation of the winners using online learning for the map size 30×30. Symbol + means won game, symbol − lost game, symbol ± means that the winner represents both won and lost games.

Graphical representation of active neurons and their association to the won or lost games (Fig. 6.4) shows the ability of the map to categorize the inputs of the game. The map is successful in inputs differentiation into clusters of the same result without knowing rules of the game (15 neurons out of 119 represent both won and lost games). There are multiple clusters that are not clearly differentiated but there are no clusters of opposite values lying next to each other.

Batch learning

Map size of 10×10 achieves worse results in case of number of active neurons using batch learning. There is maximum of 5 active neurons in the case of 500 epochs. The resulting trained map (Fig. 6.5) shows the reason for the low number of the active neurons. Components are divided into more and less active regions. In spite of the components having clear differentiation from highest values to lowest values, the result is a very low number of active neurons.

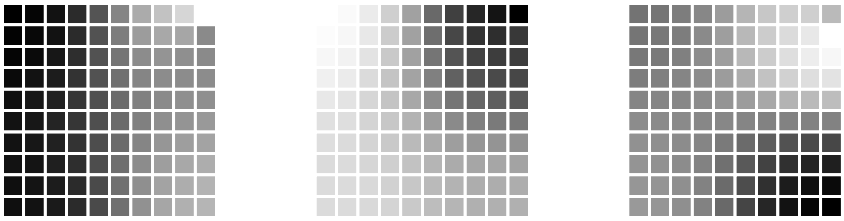

Figure 6.5: The first three components of weight vectors of the trained map 10×10 over 500 epochs using batch learning. For every neuron his first (second, third) component of weight vector is shown.

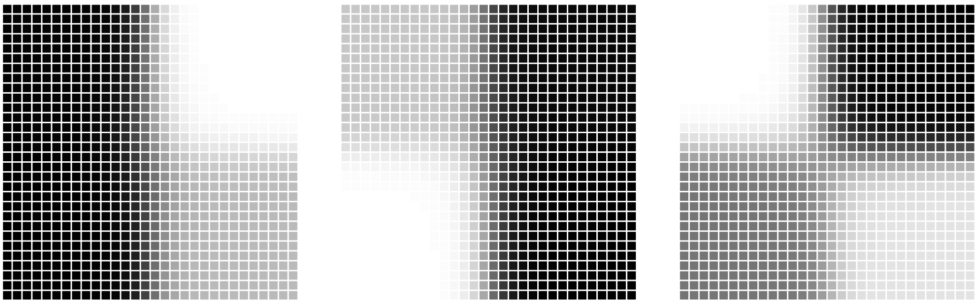

Figure 6.6: The first three components of weight vectors of the trained map 30×30 over 500 epochs using batch learning. For every neuron his first (second, third) component of weight vector is shown. Components are clearly differentiated into regions.

For one particular neuron the distance between the weight vector and the input is high even though partially (the first three components) the distance is low. This result is likely when using batch learning where multiple winners influence the weight vector of one neuron and so the resulting weight vector is an average over different inputs. For this data set the inputs are different in at least 3 values of the vector size of 27.

Increasing the size of the map to 15×15 (over 400 epochs) and also to 20×20 (over 500 epochs) did not help for this data set using batch learning to increase the number of active neurons. Active neurons are located in the corners for all sizes of the map. The next increase of the map size to 25×25 neurons (over 500 epochs) brought three times better results than in previous cases.

For the largest map (30×30) trained over 500 epochs batch learning had only 30 active neurons. For this map the first three components of weight vectors of all neurons are shown (Fig. 6.6). Although similar to online learning in the fitting of components, the structure is divided into only four areas.

Graphical representation of the winners (Fig. 6.7) shows where are all

30 winners located. As in the case of a smaller map active neurons are all located in the corners. These are not distinguishable between won and lost games. This is the result of a little number of active neurons to the data set size even though there are only 2 types of game result.

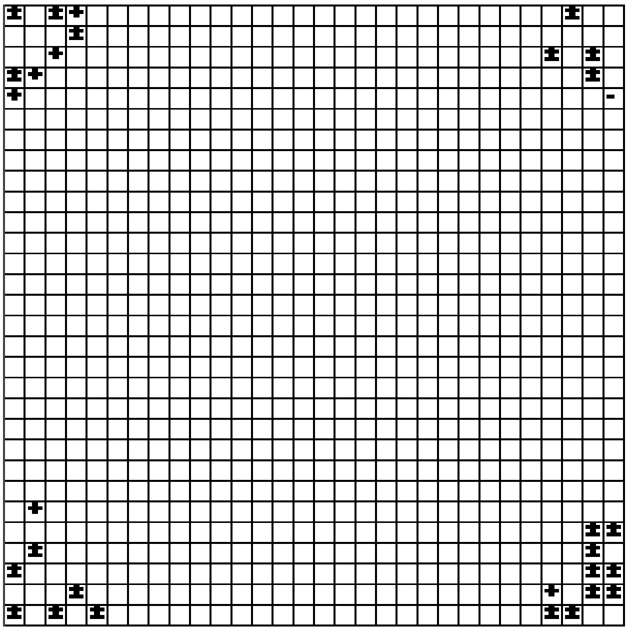

Figure 6.7: Graphical representation of the winners (BMUs) using batch learning for the map size 30×30. Symbol + means won game, symbol − lost game, symbol ± means that the winner represents both won and lost games.

Comparison of results

Comparing of the two learning algorithms on the selected data set shows the unsuitability of batch learning. The measures $\mathrm{WD}_{\mathrm{map}}$ (Tab. 6.1) and WD (Tab. 6.2) show the difference between both learning algorithms.

Even for the smallest map size the results are very different for both learning algorithms. Online learning has better results for both numerical measures ($\mathrm{WD}_{\mathrm{map}}$ and WD) on the selected data set. Comparison of the trained maps shows the difference between the learning algorithms behavior.

The difference between the results is influenced by the method of learning only. Batch learning uses result averaging from the whole epoch. In the case of big differences between input vectors batch learning algorithm can cause preferring small number of neurons in the map that cover most of the inputs from the data set. Increasing the number of epochs does help only a little to increase the number of active neurons.

Map size	online	batch
10×10	10.0	5.0
15×15	40.0	2.2
20×20	24.0	1.3
25×25	25.5	2.4
30×30	13.2	3.3

Table 6.1: $\mathrm{WD}_{\mathrm{map}}$ measure (in %) for all tested map sizes for both learning algorithms.

Map size	online	batch
10×10	4.0	0.5
15×15	5.6	0.5
20×20	10.6	0.5
25×25	10.9	1.6
30×30	12.4	3.1

Table 6.2: WD measure (in %) for all tested map sizes for both learning algorithms.

Similar results can be acquired with batch learning on animal data set (Ritter and Kohonen, 1989). The same type of encoding (localist) is used for encoding properties of 16 animals. The results of the experiment showed the ability to uniquely represent all 16 animals (map size 10×10). On the other hand, batch learning had at most 6 active neurons uniquely encoding only one animal.

Trained maps demonstrate where batch learning is different from online learning. Components of trained maps weight vectors using batch learning show map divided into four parts (three different areas). Highly active areas, middle active areas and not active areas. There are 27 components and therefore the computed distance over 27 values show similar results for very different input vectors. This problem occurs when localist input encoding is chosen. In this case, the trained map is differentiated into active and nonactive areas. Batch learning has no means to get out of local minimum (Fort et al., 2001) and therefore longer learning time increases only a little the number of active neurons. Online learning can get out of local minimum as weight vectors are dynamically shaped by learning during one epoch. This thoughts are not supported by mathematical equations yet.

We showed problems with using batch learning on the practical example. With localist encoding batch learning is less effective, takes longer and has worse results than online learning. Averaging of the results collected throughout the data set is the problem of batch learning algorithm. Deeper theoretical analysis supported by practical tests should show the precise reasons for the problem of batch learning for SOM model. Changes in batch learning algorithm are needed to accommodate the disadvantages so that it can be used for all data sets and all types of encoding. As the problem arises from the fact that input data are averaged, one proposed untested solution is to use the Eq. 6.1 not as the new weight but only as the Δ weight added to the old weight. However this solution slows down convergence of this algorithm compared to original batch learning.

Although batch learning has problems with localist encoding it is very effective in every other type of encoding. As mentioned earlier convergence of batch learning is faster, learning one epoch is faster as well and therefore batch learning should be used whenever possible. Also batch learning can be implemented as a distributed algorithm and so multiple computers can be used to compute large maps.

6.3 Batch recursive self-organizing maps

The theoretical possibility to learn recursive SOM models using batch algorithm exists as there is batch SOM algorithm as well as batch recurrent feedforward network algorithms. Different recurrent SOM models may have problem using the same approach but as there is universal taxonomy for all the existing models used (SOMSD, MSOM, RecSOM) they should have similar batch learning algorithm.

There are in principle two types of batch learning for structured data – input wise and epoch wise[2]:

- input wise batch learning (IWBL) updates weights at once for the whole input – one structure (analogy to classic BPTT learning algorithm)
- epoch wise batch learning (EWBL) updates weights after all inputs have been presented (epoch wise BPTT algorithm, batch SOM)

6.3.1 Input wise batch learning

As an inspiration to the input wise batch learning (IWBL) back propagation through time can be used. The idea of the BPTT learning algorithm (used in simple recurrent networks) is to unfold the network in history and train

[2] For non-structured data there exists only epoch wise batch learning as the input wise learning is one step even in case of online learning.

it using backpropagation learning algorithm. The same idea can be used for more complex recurrent SOM models. This way IWBL for recursive SOM can be created. The main similarity between BPTT and IWBL is the collection of the results throughout presentation of the input and updating weights after the presentation of the input. The differences between BPTT and IWBL for the recursive SOM are:

1. Output of the recursive SOM is not the actual output (which is not defined) but the output of a context function (coordinates of the winner for SOMSD, merged context and input weights for MSOM and 'output' for RecSOM),

2. The algorithm has to take the structure of the input into consideration in case of more complex structures,

3. An update has to be computed during learning just like in batch SOM (and hence the time efficiency is only slightly better than that of online learning),

4. After the last input presentation (end of a sequence, root of a tree) the map is not unfolded in time as in BPTT but simply updated the same way as batch SOM.

For every input and every vertex of the input, the winner is computed according to the model used. The weight changes are not computed at that point but only the information about the winner for current input is saved. The actual learning is similar for the input weights as in case of batch SOM (Eq. 6.6) and is similar for the context weights (Eq. 6.7) as well:

$$\mathbf{w}_i = \frac{\sum_{t=1}^{M} h(i,i^*)(t)\mathbf{x}(t)}{\sum_{t=1}^{M} h(i,i^*)(t)} \tag{6.6}$$

$$\mathbf{c}_i^{(j)} = \frac{\sum_{t=1}^{M} h(i,i^*)(t)\mathbf{q}_j(t)}{\sum_{t=1}^{M} h(i,i^*)(t)} \tag{6.7}$$

In the equations 6.6 and 6.7 M corresponds to the number of vertices in the input structure. For the context weight update index j is referring to all contexts (one for sequences, k for trees with arity k).

The space complexity slightly increases as additional memory is needed for intermediate results to be saved:

$$S(\text{rSOM}^{IWBL}) = S(\text{rSOM}) + O(M_{\max}) \tag{6.8}$$

where rSOM denotes any presented recursive SOM model and $M_{\max}$ the maximum value of M (structured input size). The space complexity will be higher only if $O(M_{\max}) > S(\text{rSOM})$ which can happen for very complex input structures.

Lower time complexity is expected for batch learning. Although there is no computation of the new weights for one input, the time complexity is asymptotically the same compared to online learning for all models as can be seen in equations for the models (Eq. 3.3, Eq. 3.8 and Eq. 3.14).

The weights are updated (Eq. 6.6 and Eq. 6.7) after the whole structure has been presented:

$$T(\text{rSOM}^{IWBL}_{epoch}) = T(\text{rSOM}^{IWBL}_{input}) + T(\text{weight change}) \tag{6.9}$$

As can be seen from the equations for all recursive models the time complexity of the weight change is asymptotically the same as for the input. Therefore the result is asymptotically the same:

$$T(\text{rSOM}^{IWBL}_{epoch}) = T(\text{rSOM}_{epoch}) \tag{6.10}$$

Also the resulting time complexity of batch learning is asymptotically the same as for online learning:

$$T(\text{rSOM}^{IWBL}) = T(\text{rSOM}) \tag{6.11}$$

The time complexity is not asymptotically better than online learning algorithm (even though for every recursive model the learning algorithm is different) and also the distributed computing is not available in this type of batch learning as the computation through structure can not be separated. For the computation of the successor all predecessors have to be already computed and saved. Therefore IWBL has only the advantage of faster convergence.

To achieve all of the above mentioned points the following scheme of IWBL can be drawn (Fig. 6.8). Pseudocode of the learning algorithm for one input (Alg. 6.1) shows the learning process in a simplified way.

Input wise batch learning is not very practical but is used for the more useful recursive SOM models batch learning algorithm – epoch wise batch learning.

6.3.2 Epoch wise batch learning

Epoch wise batch learning (EWBL) is an upgrade of IWBL algorithm. The principles are the same but the EWBL is one step further than IWBL. This algorithm is a batch algorithm for the whole epoch.

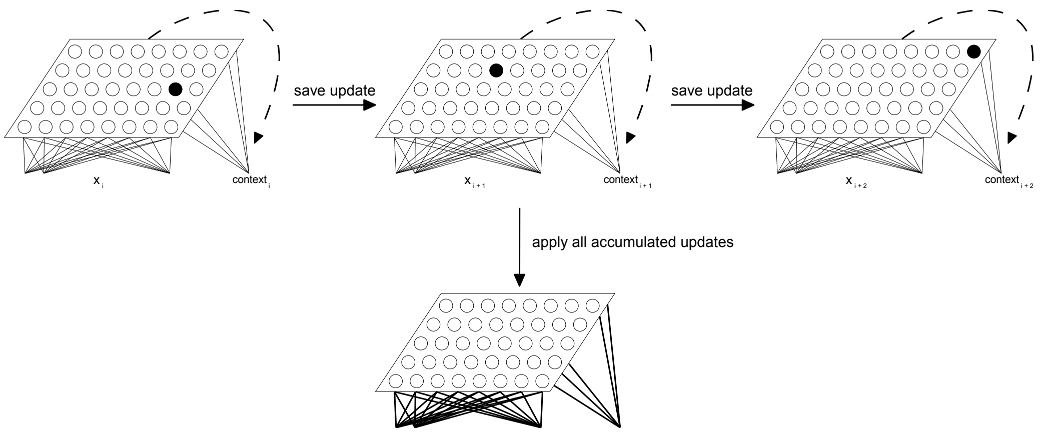

Figure 6.8: Scheme of input wise batch learning of SOM. Learning consists of two parts: the first part (top row) is the presentation of input (all vertices) with saving the state of the map and the second part is the weight change according to the vertex and the corresponding saved state.

The equations used for IWBL (Eq. 6.6 and 6.7) are also used in this learning algorithm without change. The only change is in the learning algorithm. The values of numerators and denominator are not reset after presenting every input and neither weights are changed. Instead, the numerators and the denominator are reset at the beginning of the epoch and then computed throughout the epoch on all inputs (and all their vertices). The weights are updated at the end of the epoch (after the whole data set has been presented). Pseudocode 6.2 of the EWBL shows the simplified learning algorithm.

EWBL algorithm is more useful as now the distributed computing is available for the training of the recursive SOM models. The algorithm enables the data set to be split into inputs (or input groups) that are trained separately. For every epoch the current weights (input and context) and inputs have to be provided to the slave computers which send their resulting numerators and denominator to the master computer. The master computer then sums all variables from the slave computers and computes the new weights.

This enables to work with large maps (millions of neurons) in a reasonable time even for very time consuming RecSOM model. The time complexity is not significantly lower but the distributed computing possibility makes this method very useful in real life scenarios. The online versions of the recursive SOM learning algorithms can be run only on one computer with no possibility to use processing power of more computers.

With regard to the criticism of batch learning for SOM this does not apply in this case. The reason for the exclusion is that the argument lies in localist

Algorithm 6.1 Pseudocode of the input wise batch learning algorithm for recursive SOM models. The learning algorithm is a merge of batch SOM learning algorithm and BPTT algorithm for simple recurrent networks.

```
for all neurons do
    input numerator ← 0;
    context numerators ← 0;
    denominator ← 0;
end for
for all vertices do
    input ← current vertex;
    contexts ← current contexts;
    compute winner;
    for all neurons in the map do {Eq. 6.6 and Eq. 6.7}
        input numerator ← input numerator + $h(i, i^*)\mathbf{x}(t)$;
        context numerators ← context numerators + $h(i, i^*)\mathbf{q}_j(t)$;
        denominator ← denominator + $h(i, i^*)$;
    end for
end for
for all neurons do
    new input weight ← input numerator / denominator;
    new context weights ← context numerators / denominator;
end for
```

encoding. In case of using recursive SOM models and localist encoding, the context (as part of input) will not be in localist encoding thus rendering the argument meaningless. The only model whose contexts use localist encoding is SOMSD. For this model the reasoning does not apply and in theory is not immune to localist encoding problem of SOM's batch learning.

6.4 Summary

In this chapter we presented existing batch learning algorithm for SOM and showed its advantages. However batch learning in its current form has its disadvantages too as we demonstrated in experiment with real data. The disadvantage focuses on the problem with the algorithm when using localist input encoding. We proposed update of batch learning algorithm to solve this problem.

For the recursive models we proposed two types of batch learning algorithms. Both are based on existing batch algorithms for other neural network

Algorithm 6.2 Pseudocode of the epoch wise batch learning algorithm for recursive SOM models. The learning algorithm is updated input wise batch learning algorithm expanded on the whole data set.

```
for all neurons do
    input numerator ← 0;
    context numerators ← 0;
    denominator ← 0;
end for
for all inputs do
    for all vertices do
        input ← current vertex;
        contexts ← current contexts;
        compute winner;
        for all neurons do
            input numerator ← input numerator + $h(i, i^*)\mathbf{x}(t)$;
            context numerators ← context numerators + $h(i, i^*)\mathbf{q}_j(t)$;
            denominator ← denominator + $h(i, i^*)$;
        end for
    end for
end for
for all neurons do
    new input weight ← input numerator / denominator;
    new context weights ← context numerators / denominator;
end for
```

models. The first one focuses on batch processing of one structured input. The data set is then processed similarly to SOM model with structured input behaving as one input. The second algorithm enhances the first one to process the whole data set in a batch mode. That means the actual learning is done only after the whole data set has been presented. The advantages of the mentioned batch learning for SOM apply and therefore this algorithm can be used for distributed computing of the recursive SOMs.

Chapter 7

Data extraction and reconstruction

Since the encoded tree structures become distributively encoded (approximated) by a recursive SOM, we can think of it as a distributed memory. To be able to use recursive SOM models as a memory model, the encoded data has to be extracted at least partially from the trained map[1]. The RAAM model already consists of the decoder part so no extra work is needed to extract data from the hidden layer. There is no decoder similar to RAAM model so it has to be proposed how to extract data back from the trained map.

In case of tree structures the situation is more complex. The memory depth is usually very shallow, only in the simplest cases the map memory depth is sufficient enough (Vančo and Farkaš, 2009) to be used straightforward as a memory. The data has to be reconstructed during the input presentation. We will focus on the methods of the data extraction (decoding) from the trained map if memory depth is sufficient and continue with data reconstruction if the memory depth is not as deep as to cover all data structures.

7.1 Lookup table

The simplest non-connectionist approach to data decoding from the trained map is a lookup table. If the trained map can uniquely identify the input from the state of the map (winner or map activation) then a simple lookup

[1] The applications other than memory, such as data mining or visualization, do not need this

input	winner	winner position
(d(a(a(an))))	95	[9, 5]
((dn)v)	2	[0, 2]
(an)	40	[4, 0]
(p(dn))	83	[8, 3]
((dn)(p(dn)))	3	[0, 3]
((dn)(v(d(an))))	4	[0, 4]
(a(an))	94	[9, 4]
(d(an))	96	[9, 6]
(v(dn))	93	[9, 3]
((d(an))(v(p(dn))))	5	[0, 5]
(a(a(an)))	90	[9, 0]
(v(d(an)))	80	[8, 0]
(p(d(an)))	70	[7, 0]
(dn)	51	[5, 1]
(v(p(dn)))	91	[9, 1]
a	39	[3, 9]
d	9	[0, 9]
n	49	[4, 9]
p	48	[4, 8]
v	59	[5, 9]

Table 7.1: The sample lookup table for SOMSD and map size of 10×10.

table created at the end of the training can identify input based on the state of the map.

The lookup table can have two columns, for example using the binary data set from Chapter 4 and SOMSD model with a 10×10 map (result of an experiment) can be seen in Tab. 7.1.

Using this table is very simple. After presenting one input, the state of the map is read and then the lookup table is searched. If the state is found in the lookup table the input is read in the same row. If the state is not found (that means the input was not presented during learning) the return argument has to be defined. For example the algorithm can return the closest input to the state of the map.

The possibility of implementation of the lookup table varies from a simple program memory for simple data sets to a database table for very large data sets (which means large map as well as the need to satisfy the condition that every input is uniquely identified).

The problem is that the lookup table has to be built for every training anew. This is the problem with all decoding algorithms as new training (with the randomly initialized weights) creates different weights and therefore different states of the map for the same input. For faster access the table can be sorted or in case of databases indexed.

7.2 Feedforward network as a decoder

The connectionist approach is to create a neural network on top of the trained map to work as a decoder. This is a cleaner but also a more difficult solution (the whole structure of the encoder-decoder will be a neural network). The simplest connectionist approach is to use a feedforward network. The input layer can be a trained map with activations of the current input or simply an encoded number of the winner (in case of SOMSD, MSOM, etc.). In both cases the feedforward network on top of the map will have to be trained as an auto-associator, i.e. the input to the map will be the required output. This can be either trained after every vertex (ideal for data reconstruction) or after the whole input is presented. In the latter case the requested output will be the whole input. Depending on the size of the data set, the requested output can be coded as a localist (for small data sets) or a distributed (for large data sets) vector. In Figure 7.1 an example architecture can be found over SOMSD model.

For the simplest cases only one-layer feedforward network can be used. Every neuron in the layer can be trained for one input. In other cases every neuron can have its receptive field defined and use it to decode parts of the map. These are only ideas and not implementation proposals. The implementation depends on the requirements of the decoder.

The idea of using a neural network on top of the map as a decoder can be extended. The requested output may not be the input itself but an input class for example. Activations of the trained map can serve as inputs to another module.

Another connectionist approach is to replace lookup table with a neural network model called counter-propagation network (CPN, Hecht-Nielsen (1987)). This model can be used instead of feedforward network on top of a trained map. It uses auto-association for training with hidden layer trained the same way as the SOM model.

7.3 Data reconstruction

Data reconstruction process is needed for decoding of tree as well as of long sequences. This is the case when memory depth is lower than the depth

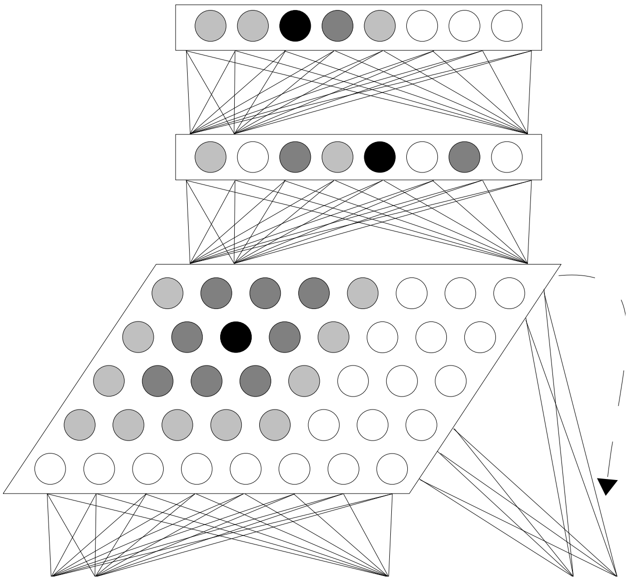

Figure 7.1: Feedforward network as a decoder over SOMSD model. In this example the feedforward network has two layers. The requested output is the whole structured input after the root has been presented to the trained map.

of the structures in the data set. In the previous chapter we introduced a measure called the tree receptive field (Section 4.1). This measure quantifies the memory depth of every neuron in the map and is higher for sequences and lower for tree structures. The quantizer depth measure shows an average memory depth and for the tree structures. It is usually around one and even below one for very large data sets (see Chapter 4).

To be able to reconstruct all structures from the trained map to their full depth, all types of receptive fields of the data set have to be found in the map. In addition the receptive field size has to be at least the size of the arity plus one. In case of sentences this size has to be at least two. The principle is to build up the structure on the fly.

That means that we do not wait for the last vertex (root) to be presented but decode the structure right after presentation of every vertex in the structure. As the presentation proceeds bottom up and data reconstruction is top down, all information about the map (usually encoded in the winner or in the case of RecSOM in the activity of the map) has to be saved for reconstruction. The reconstruction then takes the information about the map and decodes it using a chosen decoding algorithm. As only partial information is provided for every vertex, the missing information has to be provided to the decoding algorithm. This has to build the structure based on the winner's receptive field and the winner sequence. As mentioned earlier, the decoding algorithm can be a neural network or a simple algorithm. The difference is that the result from the decoding algorithm has to be saved and then the results have to be constructed to create the full structure.

As an example, let's take the data set consisting of tree structures of arity 2 (binary trees) – we can use the binary data set from Chapter 4 again. Let two trees '(a(an))' and '(a(a(an)))' have the same winning neuron in SOMSD model (thus making receptive field of this neuron to be '(a(ax))'), let it be 90 (position [9, 0]). Let the other inputs have the same winning neurons as in section 7.1 Tab. 7.1. That means that the trained map can not distinguish between the two selected trees. The processing will proceed for input '(a(an))' as follows:

- vertex 'n', no contexts, saved winner is: 49
- vertex 'a', no contexts, saved winner is: 39
- no vertex, saved winner is: null
- no vertex, saved winner is: null
- vertex with no label, contexts of winners 49 and 39, saved winner is: 40
- vertex 'a', no contexts, saved winner is: 39
- vertex with no label, contexts of winners 40 and 39, saved winner is: 90

The reconstruction process starts right after presenting the root. The sequence of saved winners is (from root to the bottom):

90, 39, 40, null, null, 39, 49.

Using the lookup table we have following receptive fields:

'(a(a -))', 'a', '(an)', null, null, 'a', 'n'

where '-' sign means unknown value.

This will be reconstructed to '(a(an))'. A similar result can be achieved using a feedforward network.

This approach only uses the receptive field information. However, in case of very large data set and therefore shallow TRF, partial information can be extracted using not only TRF but also STRF. Data reconstruction will not be complete but as STRF $\geq$ TRF it can be used to provide more information even though data in the vertices is not known. Using both data and structural information, even partial, helps in data reconstruction.

7.4 Summary

In this chapter we presented a few ideas how to use recursive SOM models as memory, i.e. how to use trained map to retrieve the original structured data. We showed universal approaches for retrieval in two cases: information can be completely retrieved from the trained map or only partial information can be retrieved from the trained map.

If the structured data can be successfully retrieved from the trained map we provided two approaches on how to do it. The first one was algorithm using lookup table that is created at the end of learning. This approach is precise but how to proceed when input is not found has to be defined based on requirements of the task. The second one was neural network approach using feedforward network on top of the map. The network can be trained during or after the training of the map. The results are not perfectly precise but the advantage of this solution is that the feedforward network can approximate the result.

For the other case when the trained map is not able to successfully encode all input data we proposed data reconstruction idea that uses receptive fields and structure receptive fields of neurons in decoding data. Also for this idea the reconstruction process runs throughout the presentations of all structured data inputs.

Chapter 8

Conclusion

In the thesis we focused on processing tree structured data with recursive self-organizng maps. The structures that can be processed are limited to trees and acyclic oriented graphs. We evaluated time and space complexity for all models to facilitate th comparison of their properties. The models process structured data differently based on the type of their context representation (feedback).

We focused on three models, SOMSD, MSOM and RecSOM. SOMSD uses coordinates of the last winner for feedback. MSOM uses merged information about the last winner, namely its context and input weights, for feedback. RecSOM, with the highest time and space complexity, uses the activation of the whole map for the feedback.

For MSOM we argued that despite the fact that the computation of the context is commutative with respect to children Hammer et al. (2004b) it does distinguish between the branches of a tree. We also added an experiment to support our theoretical claim. This makes MSOM suitable for processing trees.

To show how the selected models process tree structured data with various degrees of complexity, three data sets were chosen for experimental comparison. For this, we introduced four and applied two existing quantitative measures for this purpose. With respect to content-and-structure memory depth (TQD measure), there is no clear winner for all three data sets. Regarding the structure depth (STQD measure), SOMSD yields the best results suggesting that it can cluster trees very well according to their structural properties. The models differ in the way how they differentiate among the trees and cluster them. For SOMSD, the tree structure is more important than the content (of the labels), but the content also plays a role when the structure is the same. MSOM clusters the trees in the map more preferably by the content than SOMSD but the structure is also very important. Rec-

SOM creates complex organization in representing trees based on both the structure and the content. Regarding the uniqueness of output representations, it turns out that for the purposes of input discrimination, the winner index is only sufficient in the case of simple tree data sets, when the number of map units is higher than the number of different vertices.

We provided practical results of recursive SOMs on two XML encoded data sets, with different purposes. We presented visualization capabilities of the models on the first data set containing example library in XML format. Data mining capabilities of the recursive models were tested on the second, more complex data set, that contained information about the articles. XML format is a native format of many real life applications and we successfully implemented recursive models to use this format.

We developed batch learning for the recursive SOMs to be used more effectively. They can be implemented using multiple computers in a distributed way. This is a practical solution that enables to compute larger maps than before in a shorter time by distributing the computation. We also showed the problems with batch learning for the classic SOM and we proposed solutions to circumvent this problem.

The recursive SOMs can also be used as a memory for structured data. We provided solutions for data extraction from the trained maps, the lookup table and feed forward network, to be used for memory retrieval. In the case of shallow memory depth we proposed a data reconstruction algorithm that uses receptive fields and structure receptive fields of neurons to build up the structure from the map state.

For future work the implementation of distributed recursive SOMs is required to test the models on large data sets and large map sizes. Making the batch versions more effective is a great challenge as well. Combined with XML data processing this can lead to data mining, clustering and visualization of the whole databases. Also further extending recursive models to all graphs would mean a big step forward in processing structured data using neural networks.

Bibliography

Bar-Joseph, Z., E. Demaine, D. Gifford, and T. Jaakkola (2001). Fast optimal leaf ordering for hierarchical clustering. *Bioinformatics 17*, 22–29.

Barreto, G., A. F. R. Araújo, and S. C. Kremer (2003). A taxonomy of spatiotemporal connectionist networks revisited: The unsupervised case. *Neural Computation 15*(6), 1255–1320.

Beňušková, Ľ. (2000). Neurón a mozog. Prednáška Neurovedy I, 26.10.2000 v rámci celo-UK predmetu Kognitívne vedy.

Čerňanský, M. and P. Tiňo (2007). Comparison of echo state networks with simple recurrent networks and variable-length Markov models on symbolic sequences. In *17th International Conference on Artificial Neural Networks*, pp. 618–627.

Chappell, G. J. and J. G. Taylor (1993). The temporal Kohonen map. *Neural Networks 6*, 441–445.

Cheng, Y. (1997). Convergence and ordering of Kohonen's Batch Map. *Neural Computation 9*(8), 1667–1676.

Čihák, R. (2004). *Anatomie 3.* Grada Publishing.

Craik, F. I. and R. S. Lockhart (1972). Levels of processing: A framework for memory research. *Journal of Verbal Learning and Verbal Behavior 11*, 671–684.

Craik, F. I. and E. Tulving (1975). Depth of processing and the retention of words in episodic memory. *Journal of Experimental Psychology: General 104*(3), 268–294.

Diestel, R. (2005, August). *Graph Theory (Graduate Texts in Mathematics).* Heidelberg: Springer.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science 14*(2), 179–211.

Farkaš, I. and M. Pokorný (2007). Processing tree-structured data with the linear RAAM neural network. Technical Report TR-2007-011, Comenius University in Bratislava.

Farkaš, I. and P. Vančo (2007a). Spracovanie postupností symbolov pomocou rekurzívnych neurónových máp. *Kognice a umělý život 7*, 99–105.

Farkaš, I. and P. Vančo (2007b). Spracovanie postupností symbolov pomocou rekurzívnych neurónových máp. *Kognícia a umelý život 7*, 99–106.

Fodor, J. A. and Z. W. Pylyshyn (1988). Connectionism and cognitive architecture: a critical analysis. In S. Pinker and J. Mehler (Eds.), *Connections and Symbols*. Cambridge, Mass.: MIT Press.

Fort, J. C., M. Cottrell, and P. Letremy (2001). Stochastic on-line algorithm versus batch algorithm for quantization and self organizing maps. *Neural Networks for Signal Processing 9*, 43–52.

Fort, J. C., M. Cottrell, and P. Letremy (2002). Advantages and drawbacks of the Batch Kohonen algorithm. *ESANN Proceedings 2002*, 223–230.

Frasconi, P., M. Gori, A. Kuechler, and A. Sperduti (2001). *A Field Guide to Dynamic Recurrent Networks*, Chapter From Sequences to Data Structures: Theory and Applications, pp. 351–374. Cambridge, CA: IEEE Press.

Frasconi, P., M. Gori, and A. Sperduti (1998). A general framework for processing of data structures. *IEEE Transactions on Neural Networks 9*(5), 768–786.

Gori, M., M. Mozer, A. C. Tsoi, and R. Watrous (1997). Special issue on recurrent neural networks for sequence processing. *Neurocomputing 15*(3–4), 181–182.

Hagenbuchner, M., A. Sperduti, and A. C. Tsoi (2003). A self-organizing map for adaptive processing of structured data. *IEEE Transactions on Neural Networks 14*(3), 491–505.

Hagenbuchner, M., A. Sperduti, and A. C. Tsoi (2005a). Contextual processing of graphs using self-organizing maps. In *13th European symposium on Artificial Neural Networks*, pp. 399–404.

Hagenbuchner, M., A. Sperduti, and A. C. Tsoi (2005b). Contextual self-organizing maps for structured domains. In *ECML Workshop on Relational Machine Learning*, pp. 46–55.

Hagenbuchner, M., A. Sperduti, A. C. Tsoi, F. Trentini, F. Scarselli, and M. Gori (2005). *Lecture Notes in Computer Science 3977*, Chapter Clustering XML documents using self-organizing maps for structures, pp. 481–496. Springer-Verlag.

Hammer, B. (2000). *Learning with Recurrent Neural Networks.* Springer Lecture Notes in Control and Information Sciences 254. Springer.

Hammer, B. (2003). *Perspectives on learning symbolic data with connectionistic systems*, Chapter Adaptivity and Learning, pp. 141–160. Springer.

Hammer, B. and B. J. Jain (2004). Neural methods for non-standard data. In M. Verleysen (Ed.), *European Symposium on Artificial Neural Networks*, pp. 281–292. D-side Publications.

Hammer, B., A. Micheli, A. Sperduti, and M. Strickert (2004a). A general framework for unsupervised processing of structured data. *Neurocomputing 57*, 3–35.

Hammer, B., A. Micheli, A. Sperduti, and M. Strickert (2004b). Recursive self-organizing network models. *Neural Networks 17*(8–9), 1061–1085.

Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation.* New York: Prentice Hall.

Hebb, D. O. (1949). *The Organization of Behavior.* New York: John Wiley & Sons Inc.

Hecht-Nielsen, R. (1987). Counterpropagation networks. *Applied Optics 26*(23), 4979–4983.

Jaeger, H. (2001). Short term memory in echo state networks. Technical Report GMD Report 152, German National Research Center for Information Technology.

Jedlička, P. (2002). Synaptic plasticity, metaplasticity and BCM theory. *Bratislavské lekárske listy 103*, 137–143.

Jordan, M. I. (1989). Serial order: A parallel distributed processing approach. In *Advances in Connectionist Theory: Speech.* Hillsdale: Erlbaum.

Kaski, S., J. Kangas, and T. Kohonen (1998). Bibliography of self-organizing maps: papers: 1981-1997. *Neural Computing Surveys 1*, 102–350.

Kc, M., M. Hagenbuchner, A. C. Tsoi, F. Scarselli, A. Sperduti, and M. Gori (2006). XML document mining using contextual self-organizing maps for structures. In *INEX*, Volume 4518 of *Lecture Notes in Computer Science*, pp. 510–524.

Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological Cybernetics 43*, 59–69.

Kohonen, T. (1984). *Self-Organization and Associative Memory.* Berlin: Springer.

Kohonen, T. (1990). The self-organizing map. *Proceedings of the IEEE 78*(9), 1464–1480.

Kohonen, T. (1992). *Symp. On Neural Networks; Alliances and Perspectives in Senri.* Osaka, Japan: Senri Int. Information Institute.

Kohonen, T. (2001). *Self-organizing maps* (3rd ed.). Berlin: Springer.

Kohonen, T. and P. Somervuo (1998). Self-organizing maps of symbol strings. *Neurocomputing 21*(Issues 1–3), 19–30.

Koskela, T., M. Varsta, J. Heikkonen, and K. Kaski (1998a). Temporal sequence processing using Recurrent SOM. In *Proceedings of the 2nd International Conference on Knowledge-Based Intelligent Engineering Systems*, pp. 290–297.

Koskela, T., M. Varsta, J. Heikkonen, and K. Kaski (1998b). Time series prediction using recurrent SOM with local linear models. *International Journal of Knowledge-Based Intelligent Eng. Systems 2*(1), 60–68.

Kröse, B. J. A. and M. Eecen (1994). A self-organizing representation of sensor space for mobile robot navigation. *Proc. IROS '94 1*, 9–14.

Kurz, A. (1992). Building maps for path-planning and navigation using learning classification of external sensor data. *Artificial Neural Networks 1*(2), 587–590.

Kvasnička, V., Ľ. Beňušková, J. Pospíchal, I. Farkaš, P. Tiňo, and A. Kráľ (1997). *Introduction to the Theory of Neural Networks.* Bratislava: IRIS.

Lukosevicius, M. and H. Jaeger (2009). Reservoir computing approaches to recurrent neural network training. *Computer Science Review 3*(3), 127–149.

Martinetz, T. and K. Schulten (1991). A neural-gas network learns topologies. In *Proceeedings of the International Conference on Artificial Neural Networks*, Amsterdam, pp. 397–402. North-Holland.

McCulloch, W. S. and W. Pitts (1943). A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics 5*, 115–133.

Minsky, M. and S. Papert (1969). *Perceptrons.* Cambridge, Massachusetts: MIT Press.

Mori, R., Y. Bengio, and R. Cardin (1989). Speaker independent speech recognition with neural networks and speech knowledge. In *Advances in neural information processing systems 2*, pp. 218–225. Morgan Kaufmann Publishers Inc.

Návrat, P., M. Bieliková, Ľ. Beňušková, I. Kapustík, and M. Unger (2002). *Umelá inteligencia.* Vydavatelstvo STU, Bratislava.

Neubauer, N. (2005). Recursive SOMs and Automata. Master's thesis, Cognitive Science, University of Osnabrück.

Palmer, C. (2005). Sequence memory in music performance. *Current Directions in Psychological Science 14*, 247–250.

Pöllä, M., T. Honkela, and T. Kohonen (2006). Bibliography of self-organizing map (SOM) papers: 2002-2005. Unpublished SOM bibliography.

Pollack, J. (1990). Recursive distributed representations. *Artificial Intelligence 46*(1-2), 77–105.

Ritter, H. (1997). Self-organizing maps for robot control. *International Conference on Artificial Neural Networks 1327 1997*, 673–684.

Ritter, H. and T. Kohonen (1989). Self-organizing semantic maps. *Biological Cybernetics 61*, 241–254.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review 65*, 386–408.

Rumelhart, D. E., G. Hinton, and R. J. Williams (1986). *Parallel Distributed Processing*. Cambridge, MA: The MIT Press.

Šíma, J. and R. Neruda (1997). *Teoretické otázky neuronových sítí*. Matfyz Press.

Sinčák, P. and G. Andrejková (1996a). *Neurónové siete (Inžiniersky prístup)*, Volume 1. Elfa s.r.o.

Sinčák, P. and G. Andrejková (1996b). *Neurónové siete (Inžiniersky prístup)*, Volume 2. Elfa s.r.o.

Sperduti, A. and A. Starita (1997). Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks 8*(3), 714–735.

Steil, J., R. Koiva, and A. Sperduti (2006). Unsupervised clustering of continuous trajectories of kinematic trees with SOM-SD. In *Proceedings of the 14th European Symposium on Artificial Neural Networks*, Bruges, Belgium, pp. 1–6.

Strickert, M. and B. Hammer (2003). Unsupervised recursive sequence processing. In *Neurocomputing*, pp. 433–439. D-side Publications.

Strickert, M. and B. Hammer (2004). Self-organizing context learning. In *European Symposium on Artificial Neural Networks*, pp. 39–44.

Strickert, M. and B. Hammer (2005). Merge SOM for temporal data. *Neurocomputing 64*, 39–71.

Tai, W. (1995). A batch training network for self-organization. In *International Conference on Artificial Neural Networks*, Volume 2, pp. 33–37.

Tiňo, P., I. Farkaš, and J. van Mourik (2006). Dynamics and topographic organization of recursive self-organizing maps. *Neural Computation 18*(10), 2529–2567.

University of California Irvine (2009). Machine learning repository. internet. http://archive.ics.uci.edu/ml/.

Vančo, P. (2009a). Dynamika dávkového učenia na modeloch samoorganizujúcich sa máp. *Kognícia a umelý život 9*, 357–362.

Vančo, P. (2009b). Visualization of simple XML data using recursive self-organizing neural maps. *Informatics 2009 : International Conference on Informatics 10*, 341–346.

Vančo, P. (2010). Dekódovanie štruktúrovaných dát z natrénovaných rekurentných SOM. *Kognícia a umelý život 10*. Submitted.

Vančo, P. and I. Farkaš (2009). Recursive self-organizing networks for processing tree structures: Empirical comparison. In *IJCCI 2009 : Proceedings of the International Joint Conference on Computational Intelligence*, Volume 64, pp. 459–466.

Vančo, P. and I. Farkaš (2010). Experimental comparison of recursive self-organizing maps for processing tree-structured data. *Neurocomputing 73*(7–9), 1362–1375.

Voegtlin, T. (2002a). *Neural Networks and Self-Reference*. Ph. D. thesis, Universite Lyon 2.

Voegtlin, T. (2002b). Recursive self-organizing maps. *Neural Networks 15*(8-9), 979–992.

Werbos, P. J. (1990). Backpropagation through time: What does it do and how to do it. *Proceedings of IEEE 78*, 1550–1560.

Williams, R. J. and D. Zipser (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural Computation 1*(2), 270–280.