Generalized Self Organizing Maps - An object oriented framework

Diplomarbeit

zur Erlangung des Akademischen Grades
Diplom-Ingenieur
in der Studienrichtung
Technische Mathematik
(Computerwissenschaften 869)

Eingereicht von

Mario Drobics (9325139)

Angefertigt am
Institut für Symbolisches Rechnen (RISC)
der TNF - Universität Linz
unter der Betreuung von
Herrn Univ. Prof. DDr. Bruno Buchberger

Linz im Februar 2000

Special Thanks to
Edward Blurock
for his excellent support!

Contents

1 Introduction **1**
- 1.1 Motivation 2
  - 1.1.1 Self Organizing Maps 4
- 1.2 State of Approach 5
  - 1.2.1 State of the art 5
  - 1.2.2 Component analysis 6
  - 1.2.3 Implementation 7
  - 1.2.4 Examples 8
  - 1.2.5 Outlook 9

2 State of the art **10**
- 2.1 Kohonen's model 11
  - 2.1.1 Mathematical model 11
  - 2.1.2 Algorithm 15
  - 2.1.3 Analysis 17
- 2.2 Neural Gas plus Competitive Hebbian Learning 19
  - 2.2.1 Algorithm 19
  - 2.2.2 Analysis 20
- 2.3 Growing Neural Gas 20
  - 2.3.1 Algorithm 21
  - 2.3.2 Analysis 23
- 2.4 Growing Cell Structures 23
  - 2.4.1 Algorithm 23
  - 2.4.2 Analysis 25
- 2.5 Growing Grid 26
  - 2.5.1 Algorithm 26
  - 2.5.2 Analysis 27
- 2.6 Growing Hypercubial Output Space (GSOM) 27
  - 2.6.1 Mathematical model 27
  - 2.6.2 Algorithm 29
  - 2.6.3 Analysis 29
- 2.7 Common SOM objectives 29
- 2.8 Valuation measures 30

2.8.1 Topographic product 31
2.8.2 Topographic function 31

3 Component analysis **33**
3.1 Generalization 33
3.2 Component classes 36
3.2.1 Map data 36
3.2.2 Adaption 38
3.2.3 UpdateNode 38
3.2.4 Learning Rate 39
3.2.5 Neighborhood Function 40
3.2.6 AdaptionNet 41
3.2.7 Additional steps 42

4 Implementation **43**
4.1 Generalized Class structure 43
4.1.1 SOMMap Data 44
4.1.2 SOMMap operators 45
4.1.3 Basic utilities 48
4.2 Component Classes 49
4.2.1 class Graph 49
4.2.2 class Map 50
4.2.3 class SOMWeight 50
4.2.4 class SOMTrainNet 50
4.2.5 class SOMBreakCondition 51
4.2.6 class SOMAdaption 51
4.2.7 class SOMBestMatching 51
4.2.8 class SOMUpdateNode 52
4.2.9 class SOMLearningRate 52
4.2.10 class SOMNeighborhoodFunction 53

5 Extensions **54**
5.1 Class structure 54
5.1.1 Neural Gas 54
5.1.2 Growing Neural Gas 56
5.1.3 Growing Cells 59
5.2 Component Classes 59
5.2.1 class SOMAdaptionNeuralGas 61
5.2.2 class SOMBestMatchingNeuralGas 61
5.2.3 class SOMNeighborhoodFunctionNeuralGas 62
5.2.4 class SOMTrainNetGrowing 62
5.2.5 class SOMAdaptionGNG 63
5.2.6 class SOMAdaptionNetGNG 63
5.2.7 class SOMAdaptionGCells 64

5.2.8 class SOMAdaptionNetGCells 64

6 Examples 66
6.1 A simple sample 66
6.1.1 Kohonen Grid 66
6.1.2 Neural Gas 67
6.1.3 Growing Neural Gas 68
6.2 Animals 68
6.2.1 Kohonen Grid 70
6.2.2 Neural Gas 71
6.3 Iris flowers 72
6.3.1 Neural Gas 74
6.3.2 Growing Neural Gas 75

7 Outlook 77
7.1 Extensions 77
7.1.1 New algorithms 77
7.1.2 New components 78
7.1.3 Special Applications 79
7.2 Conclusion 80

A Input files 82
A.1 Map data 82
A.2 Map operators 86
A.3 Basic utilities 89
A.4 Neural Gas 92
A.5 Growing Neural Gas 93

B Test files 95
B.1 A simple sample 95
B.1.1 Kohonen Grid 95
B.1.2 Neural Gas 99
B.1.3 Growing Neural Gas 102
B.2 Animals 106
B.2.1 Animals Data set 106
B.2.2 Kohonen Grid 109
B.2.3 Neural Gas 112
B.3 Iris 115
B.3.1 Iris Data set 115
B.3.2 Neural Gas 122
B.3.3 Growing Neural Gas 125

C Notation 128

Abstract

To model a physical system on the computer, one tries to understand what parameters influence the system and how they do. This information can be gained using *Data Analysis*. Analysis may be done using statistical methods (e.g regression analysis) or *Machine Learning* methods (ML) like decision trees, cluster analysis or Artificial Neural Nets. Though all these ML methods differ in structural and algorithmic form and in how they search for a solution (in some kind an optimization process), they all try to abstract general concepts using specific examples (*inductive learning*).

Neural Nets are inspired by the human brain as they use small but highly connected components (neurons) to compute the result. Information is not stored at a single location but spread over the whole network.

Self organizing maps (a special kind of artificial neural nets) are able to map a data set onto a predefined or dynamicly created (discrete) topological structure. This ability can be used for generalization purpose (finding clusters and neighborhood relations) as well as for approximation tasks (e.g. approximating a more dimensional function with a n-dimensional grid). Though the original model developed by Kohonen uses a fix net structure, several extensions have been developed recently, which create or modify the size of the net on demand (*Growing Neural Maps*).

In this paper a general object oriented framework for neural maps will be developed. Generalization was done in two ways. First the structure of the map was generalized. This allows not only to use grids (or any other predefined structure), but any general graph. By doing an analysis of the different algorithms, the interchangeable parts were found. This allows to setup any concrete algorithm or variant, by substituting the appropriate parts.

The system is built using object oriented design. The kernel of the system is Kohonens original algorithm and a net, based upon a general graph. Using this, some extensions are described: Neural Gas, Growing Neural Gas, Growing Cells, Growing Grid and GSOMs. The general net structure is also extended to allow the setup of a n-dimensional grid. The framework was implemented within the *Analysis++* package using *C++*.

To give an overview of the abilities of the methods and to show, how different algorithms deal with the same data-set, three examples are given at the end.

Chapter 1

Introduction

In the past, data was usually "processed" by a given algorithm or formula to gain a specific result. This required detailed knowledge about the contents and structure of the given data set. As this knowledge is not always available, a new branch of methods was developed, that should allow to "analyze" new (and properly unknown) data in some way. In the beginning, *Data Analysis* was usually done using statistical methods. But these methods also required a lot of knowledge, an in-sight of the given data. With the arising of computers, and their ability to deal with great amounts of data, it was now possible to do additional analysis.

One of these new developed branches was *Machine Learning*. This means, that a *machine* (the computer), tries to find similar parts (generalization) and put different parts into relation to each other (*learning*). How learning is done, differs a lot between the different methods, but one can distinguish two approaches: *Supervised Learning*, where learning proceeds by presenting not only the input data-set, but also the desired output (e.g. if animals are described, the species), and *Unsupervised Learning*, where learning proceeds without this additional information [5, 6].

A special branch of Machine Learning methods are Artificial Neural Nets (ANN). These methods try to recreate the features of the human brain [1, 2, 3, 4]. Several different variants exist, supervised as well as unsupervised. One of the most flexible ANN methods are *Neural Maps*.

In this paper I will present an object oriented framework for *Neural Maps*. Neural Maps are able, to approximate the distribution of unknown input data and to map it onto a new topological structure. As opposed to most Neural Net methods, where learning is done by supervised learning methods, the algorithms used here support unsupervised learning.

The original model was developed by Kohonen and is known as *Self Organizing Map* or SOM [13, 16]. Though the framework is flexible enough to hold most derivations of the basic model, the focus in this paper was set on growing nets. This group of methods allows not only the mapping of the input data onto a given topological structure, but also creating this structure on demand.

The goal of this thesis is to generalize the basic algorithm. For that purpose the main components are isolated and put into relation to the the whole algorithm. This generalization is then used, to describe several extensions in their relation to the original algorithm and to each other.

The methods were implemented as part of the Analysis++ package, using an Object Oriented approach. This enables users of the methods to easily interchange parts of the algorithms and to compare the results. While most previous implementations of these algorithms only supported a special case of each algorithm, this framework is completely modifiable. Each method can be replaced to test the results with different setups. This is a great benefit, because one program can be used to do several tests using the same input data. Also developing new methods is enormous simplified, as any old method can be interchanged within the algorithm while the other methods are reused, so that one can concentrate on the new parts.

The application of these methods range from dimension reduction of high dimensional data over vector quantification to clustering. At the end of the paper some examples are given to illustrate these abilities.

The program was completely written in C++ and is part of the Analysis++ project.

1.1 Motivation

Artificial Neural Nets (ANN) try to recreate the features of the human brain. Like our brain, each net consists of a great number of neurons, which are connected with each other. Information processing is not done by taking the input data and performing a single computation to obtain the final result, but rather by spreading the information over the whole net and multiple small computations in each neuron. The neurons, that receive input information form the input layer, the neurons that contain the final result, form the output layer (see figure 1.1). Some nets use neurons that lie between this two layers. They build the so called hidden layer. The neurons are connected by directed weighted edges so that the output of a neuron in one layer is used as the input of a neuron in the next layer. An introduction and overview of ANNs can be found in [1, 2, 3, 4].

As opposed to most classical methods, the behavior of the net is not predefined, but modified as part of the learning process. For that purpose, sample data is presented to the net and then the net is adapted to improve the results. This may be done by comparing the results with the desired output data or by self-organizing methods.

As the information is spread over the whole net and each neuron has only a limited influence to the final result, ANN's (an in particular SOM's)

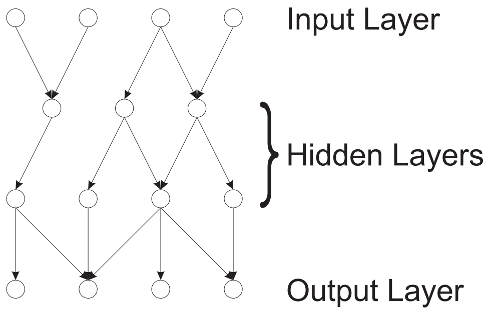

Figure 1.1: A basic ANN with input-, hidden- and output-layer

have a lot of benefits:

- Flexible

  ANN's need not to be "programmed" to fit to a special application, but are able to adapt their behavior. They are also able to learn similarities and to map the structure of the input data onto their internal one. This enables ANNs not only to do straight forward input-output matching, but also to handle unknown data according to known inputs.

  A net used to classify flowers, may also be used to find similar animals. The only differences are the parameters of the net and the input data used for training.

- Adaptive

  As learning must not stop after a fix time, some nets remain adaptive also in the application phase. For each new input sample, the net is modified. This enables the net to follow non-stationary input data and to adapt to new situations.

  For example if a robot is controlled via an ANN. After a while the bearings may wear out and the robot will have problems to position its arms correctly. In that case, the ANN is able to correct these errors, as it adapts its parameters after each positioning.

- Ability to process wrong or incomplete data

  Neural Nets are able to deal with blurred or wrong data, as they map similar inputs to similar outputs. If for example one parameter is wrong or missing, the net is still able to reproduce an according output. This is a great benefit against classical computations, where wrong input data can cause fatal errors.

- Supports massive parallel processing

The information in an ANN is spread over the whole network to a lot of neurons. Each intermediate computed by this neurons is then again spread to the next layer of neurons, until the final result is obtained. This parallel computation of the result is ideal for massive parallel computing. If special hardware is used, computations can be accelerated.

- Fault resistant

  As each neuron has only a limited influence to the final result, a defect of one neuron has also only limited effect. This is especially important if computation is done in awkward environments (e.g. beneath a furnace).

1.1.1 Self Organizing Maps

Self Organizing Maps were introduced in 1982 by T. Kohonen [12]. The motivation was to find an algorithm, "that would effectively map similar patterns (pattern vectors close to each other in the input signal space) onto continuous locations in the output space". To solve this problem, he investigated the SOM algorithm. The algorithm creates a mapping from the input space onto a discrete topological structure (e.g. a grid), where the above criteria is fulfilled. This mapping is modified in each learning step, to minimize the overall error. To achieve a continuous mapping, not only one neuron, but also its neighbors in the output space are modified. This method of lateral interaction was used before for associative memory [7, 8, 9, 10], but opposed to Kohonen's model, a continuous output space was used.

While this algorithm works unsupervised (that means that learning precedes by presenting new input samples but not the desired output), also supervised learning algorithms were developed, but will not be discussed here. Nevertheless it is possible to extend the given framework to integrate these methods. The most popular extension is LVQ (learning vector quantization) developed from Kohonen [13] where supervised learning is used for clustering. A good overview of possible derivations can be found in [16].

A generalization of Kohonen's SOMs are growing maps. While Kohonen used a fix topological structure in the output space, Growing Maps are able to create this structure on their own. Some methods are able to create a complete new structure, while other only "expand" a given one (e.g. a grid) to the needed size. This allows an easier analysis of unknown and / or very unequal distributed input data. It is also possible to use these methods for simple cluster analysis.

1.2 State of Approach

This thesis will put Kohonen maps and some growing variants in one structural framework. After a detailed description of the different algorithms, they are analyzed to find similarities and differences. This analysis is used to create a common class structure for all algorithms and possible extensions. Afterwards the implementation of these classes is described. To show the abilities of these methods, some examples are given in the end.

1.2.1 State of the art

Starting with Kohonen's SOM algorithm several extensions are discussed in chapter 2.

- Kohonen's Basic Algorithm

  Kohonen's algorithm is the basic part of all other Neural Map algorithms. Nevertheless there are a lot of possible modifications, like the definition of "similar" or the update function. It is one goal of this paper to isolate these components, to allow easy interchange and to enable the user to test different setups for a given dataset.

- Neural Gas plus Competitive Hebbian Learning

  The *Neural Gas* algorithm is an extension of classical vector quantization methods, that try to approximate the input data with a set of reference vectors. This concept is enhanced by a lateral interaction between these vectors (nearby vectors influence each other), so that not only the distribution of the input data, but additionally neighborhood relations are learned.

- Growing Neural Gas

  This model uses a flexible number of neurons to define the map. These neurons are connected, using *Competitive Hebbian Learning* as in the Neural Gas (often used connections are strengthened while rarely used ones are removed). A local error measure is used, to determine, where to insert new neurons.

- Growing Cell Structures

  This model is closely related to the *growing neural gas* model, but it uses a fix network dimension. This is done, by not only adding a new connection, but a complete simplex to the net.

- Growing Grid

  This model is a combination of ideas from the growing-cells using a classic grid as underlying structure. Growing is done, by inserting a whole row or column at the point with the highest local error measure.

- Growing Hypercubial Output Space (GSOM)

  The GSOM model is in its basic idea also a growing grid algorithm. As opposed to the Growing Grid, the GSOM can also grow into a new dimension, so that the network dimensionality has not to be known a priori.

1.2.2 Component analysis

The goal of this paper is to create a framework so that all Neural Map methods can fit in. To gain maximum flexibility, an object oriented approach was taken.

Object Orientation (OO) has become the main philosophy in program design in the last few years. This is not only a new hype, but rather reflects the need for easier and more efficient software development. OO-Design offers a lot of benefits as well for the programmer, as for the user:

- Clear design

  OO-Design requires a clear structure of the used components. Each of these components has a predefined interface and holds both, the data and the methods to manipulate it. This ensures easy readability and prohibits uncontrolled data manipulation.

- Interchangeability

  As objects communicate via messages, it is easy to interchange a component with a new one. E.g. the method that computes the distance between two points can be replaced by an other one that uses a different metric. This only affects the metric operation, while the rest of the framework (including all calls to the distance method) stays untouched.

- Reusability

  If a new class is created, it is possible to reuse previously designed classes to minimize coding. This is done via heredity of components. A grid for example, has the same interface as a graph, but differs in some methods, like adding a neuron. These methods need to be rewritten, while the rest of the graph-class can be reused.

Especially when testing different methods and creating new ones, the above properties of OO-Design are very useful. They enable the programmer to minimize new code, when creating new methods and enable the user to easily interchange methods and combine them to new algorithms as needed. As this can be controlled via the input files, results can be compared without creating new programs.

In chapter 3, a generalized algorithm will be developed, that allows easy modifications of the basic steps, so that every variant of SOM algorithms is possible to fit in it.

This was done, by dividing Kohonen's algorithm in its basic parts and then modifying and extending these parts to fit also for the other algorithms. This horizontal view allows easy comparison between the different algorithms. It is shown, that only small parts differ in each method and that great parts are interchangeable.

1.2.3 Implementation

In chapter 4 the theoretical results from chapter 3 are taken to build a concrete class structure. First the general framework is build up. Each part of the algorithm is isolated in a own class, containing all needed methods and data. Dynamic references are used to link these objects together. The class dependencies are described using UML (Unified Modeling Language) diagrams. Then the most important parts are described in detail using pseudo-code algorithms.

Implementation was done within the *Analysis++* package, which offers a C++ framekit for Machine Learning. Here the main components were isolated and implemented as C++ objects. This allows the interchange of new modules with previously developed components. This enables the user to compare the results of the methods with the results of classical algorithms. As all new methods were fit into the framework, it is possible to use the datatypes and data manipulation functions defined in the package. So not only continuous data like floats can be used as an input, but also (fuzzy-) logical values.

On the other hand, the results of the new methods may be reused in the package. As Neural Maps are able to do dimension-reduction of high dimensional input data, they may be used to preprocess such datasets before using them in other methods. This is especially useful, if the input data is very complex or the data is changing in time. In both cases, the net is able to act as a buffer, that "normalizes" the input data.

The basic components are implemented in two basic modules, one with the kernel components and virtual classes needed for all Neural Map implementations and a second module with some special utilities (e.g. a grid and a numeric vector class).

To gain maximal flexibility the map data is stored as a undirected graph. Each component has a predefined interface and is dynamically linked with the other modules. This enables the user to interchange the components even at run-time. The algorithms can be controlled through the input files. The input files can be found in the Appendix.

In chapter 5 this set of methods is expanded by some growing algorithms in a separate module. Using OO-design the new parts were easily fit into the basic model. As some models are extensions of other ones, a lot of components could be reused. This ensures, that the interface is similar and coding is minimized.

The ANALYSIS++ program is a object-oriented framework written in C++ for the manipulation and development of multi-strategy machine learning methods and data objects. The projects is being developed within an EC project, REFORM in close cooperation with UNI-SOFTWARE PLUS (USP) and the Fuzzy Logic Laboratory Linz (FLLL). The emphasis of the project is to provide a platform in which many machine learning methods can be used in harmony together, i.e. the multi-strategy approach. Two main concepts to this end are the intelligent use of interchangeable datatypes and a mechanism to store the history of the data analyses performed on a data set. Another branch of the project is concerned with the generalization of traditional methods into an object-oriented framework. Another aspect is the generalization of machine learning methods using simple true-false logic to fuzzy logics (not only as a post-processing optimization, but also in the direct creation in a fuzzy logic domain).

1.2.4 Examples

To illustrate the power and flexibility of the implemented methods, several examples are presented in chapter 6. For that purpose three different data-sets are analyzed using different methods.

- Simple sample

  As an introducing example, a simple square is presented to each algorithm. The goal is to distribute the net homogeneously over the square. This sample was run with Kohonen's original algorithm, the Neural Gas and the Growing Neural Gas algorithm. This sample shows, how the net self-organizes to approximate the input data and to prevent neighborhood relations.

- Iris flowers

  This is a classic AI sample data-set, where the goal is to separate three species of iris flowers, which are described by four discrete parameters. The result is comparable to that, obtained by classical methods, but without any class information in the learning process. This shows, how the net is able to find structures in the input data and create according nets.

- Animal sample

  Here different animals are described by several boolean attributes. Goal of the learning process is, to find relations between the animals and to group similar ones together. The data was presented to Kohonen's original algorithm and to the Neural Gas. Both create good mappings of the relations between the animals. The Neural Gas is furthermore able to create clusters of assimiable animals.

These samples give a survey of the possible applications of the presented algorithms. As the input data is handled in a very flexible way, and a lot of different information may be received form the results, these methods are ideal for dealing with unknown and complex data. Other classic applications for SOMs are speech recognition [14] or the traveling salesman problem [15].

By using one dataset with several algorithms it is also shown, how simple the behavior of the net can be changed. This enables the user to "play" with different methods, compare the results and select the best method for his needs.

1.2.5 Outlook

The paper is closed with an short outlook on possible extensions and variants of the methods implemented so far. In chapter 7 the development of new algorithm is discussed as well as the design of new components. Also several modifications that fit ANNs especially for cluster analysis are described.

In the future, the framework presented in this paper will hopefully be extended with a lot of new variants and algorithms.

Chapter 2

State of the art

In this chapter I will give a brief introduction to the main ideas of self-organizing maps. The basic concepts are explained starting with Kohonen's model, which is presented from a functional point of view and as an algorithm. The algorithm is analyzed and divided into its basic components.

Though SOMs can be described as two-layer ANNs, they also define a mapping from the input onto the output space. The output space can be any discrete topological structure. Lateral interaction is used to adjust the net. This provides a smooth global ordering of the net, so that similar inputs are mapped onto topological close regions on the net [12].

Kohonen used a predefined output structure, usually a two-dimensional grid. For some applications, this definition is to restrictive. E.g. if the input data is very high dimensional or has a lot of distinct clusters, a grid will not suffice. Therefore others have developed several extensions that create the output structure on demand. Some of them will be described here:

Neural Gas plus Competitive Hebbian Learning: The Neural Gas algorithm starts with a set of unconnected neurons. Then connections are created as an part of the learning process [22, 24].

Growing Neural Gas: This model extends the Neural Gas algorithm. Only two neurons are created at startup, while additional neurons are generated in the learning process [25].

Growing Cell Structures: To create a map with a homogeneous structure, this algorithm not just adds single neurons, but complete simplices [23].

Growing Grid: In some applications one may need the strong structure of a grid as underlying structure. For that purpose this algorithm is able to insert new rows to the grid [26].

Growing Hypercubial Output Space (GSOM): The GSOM Model is in its basic idea also a growing grid algorithm. As opposed to the

Growing Grid, the GSOM is able to grow into a new dimension, so that the network dimensionality has not to be known a priori [28].

2.1 Kohonen's model

Kohonen investigated the Self Organizing Maps (SOM) in 1982 [12]. His idea was to create a neural net to represent the input space, using the topological structure of a grid to store neighborhood relations. As opposed to most Neural Net methods, where the desired result is used to compute the weights of the net, SOM's need no reference output to tune their parameters but are able to self-organize themself.

A neural net *self-organizes* if learning proceeds without any supervision. The input data is presented to the net, and it tries to find a corresponding representation of the neighborhood relations of the input space itself. The goal of the learning process is to minimize a global error function (e.g. the average prediction error). For each new sample, the net parameters are modified to optimize this error function.
Kohonen's algorithm has two major goals:

Topology-preserving Similar input vectors should be mapped to nearby neurons of the net, and adjacent neurons should have comparable reference-vectors.

Distribution-preserving The distribution of the weight vectors should approximate the density of the input data $P(x)$.

As learning is done locally but with limited interaction, a detailed mathematical analysis of how learning is done, is difficult and still missing. An analysis how the neurons are distributed after the learning process was done by Kohonen [12] and Ritter [17].

This model is in some way a clustering method as it divide the input space in several partitions. It is closely related to the *k-means cluster analysis*, which was developed from Andersberg in 1973 [11]. The main difference is the *topological structure* that stores the neighborhood relations. This enables ANNs to be uses also for other applications then cluster analysis.

2.1.1 Mathematical model

Self Organizing Maps belong to the class of Neural Nets. Though SOMs are usually described as an independent development, I will try to introduce them as a classic neural net first.

SOMs can be described as simple two-layer networks. This means, that they consist of an input layer, where the input data is presented to the net and an output layer, where the results are displayed. These two layers are connected by weighted edges. The input space has to be a vector-space to allow computation with the input data inside the network. For the output space, any graph structure suffices.

As this representation is very limited, an alternative model will be developed afterwards, that describes the map from a more general point of view.

Layer representation

The net consists of two layers. The input or source layer $\mathcal{R}$ and the output or target layer $\mathcal{C}$. Kohonen used a grid as output structure, so that every neuron has exactly four neighbors (except the neurons on the edge). The input layer comprises N_R (this is the dimension of the input space), the output layer N_C units (the number of different output states). The neurons of the input layer are fully interconnected to the neurons of the output layer by an edge with weight w_{is} (see figure 2.1). Both, the input vectors and the weight vectors have to be normalized. This is necessary to prevent favoring of vectors with great values.

If a input vector $x = [\xi_1, \xi_2, \ldots, \xi_{N_R}]^T \in \mathcal{R}$ is presented to the net, for every neuron on the output layer $j \in \mathcal{C}$ the *energy function* $e_j(x)$ is computed:

$$e_j(x) = \sum_{i=1 \ldots N_R} \xi_i w_{ij}. \tag{2.1}$$

The most stimulated neuron $s \in \mathcal{C}$

$$s = \mathrm{argmax}_{j \in \mathcal{C}} \ \{e_j(x)\}. \tag{2.2}$$

is defined as the *best corresponding neuron* and its weights $w_{is}, \ i = 1 \ldots N_R$ are updated. Adjacent output neurons are also modified according to their distance to s. Usually the minimal path distance is used, but any other metric (e.g. the city-block metric) would suffice too.

For each input signal, the best matching neuron is defined. From a functional point of view, this relation defines a mapping from the input-space onto the output-space. Each input signal x is mapped onto the neuron s with the highest energy level $e_s(x)$. If arbitrary energy functions are used, this leads to an alternative definition of SOMs as adaptive mappings.

Map representation

The SOM is defined as a mapping of the n-dimensional input space $\mathcal{R}$ onto the m-dimensional output space $\mathcal{C}$ (mostly $m = 1$ or 2). The output space consists of N_C neurons. They are embedded in the topological structure of $\mathcal{C}$, which may be an m-dimensional grid or any other graph structure. To each neuron s, a parametric *weight vector* $w_s = [\mu_{s1}, \mu_{s2}, \ldots, \mu_{sn}]^T \in \mathcal{R}$ is associated that is part of the input space. See figure 2.2

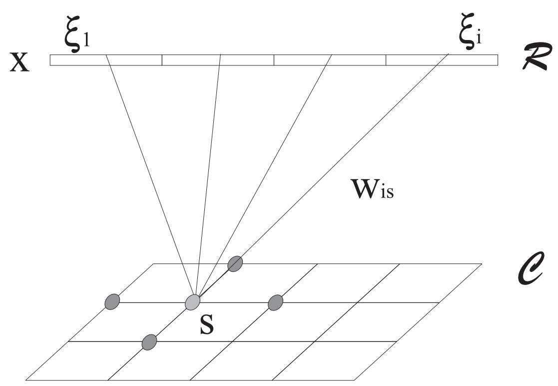

Figure 2.1: Kohonen net, Layer representation

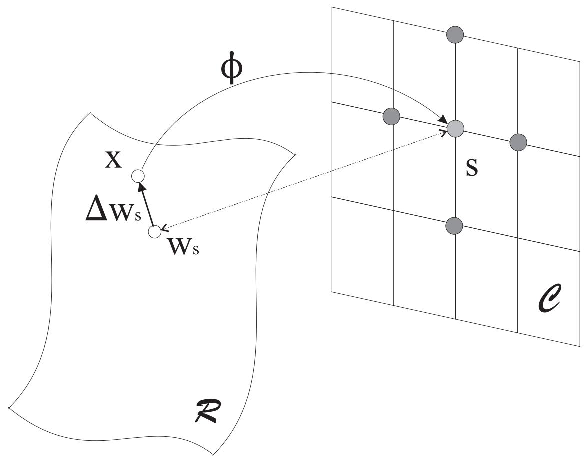

Figure 2.2: Kohonen net, Map representation

The mapping

$$\phi_w : \mathcal{R} \mapsto \mathcal{C}, x \mapsto \phi_w(x), \tag{2.3}$$

defines the *neural map* from the input space $\mathcal{R}$ to the topological structure $\mathcal{C}$. $\phi_w(x)$ is defined by

$$\phi_w(x) = \text{argmin}_{i \in \mathcal{C}} \ \{\|x - w_i\|\}. \tag{2.4}$$

This means, that every input signal is mapped onto that neuron of the output layer, whose weight vector is closest to the input. The norm used to compute this distance can be chosen arbitrarily.

This is equivalent to the computation of a "negative" energy function $-e_i(x)$ ($\text{argmin}_i e_i(x) = \text{argmax}_i(-e_i(x))$). If the dot product is used instead of the general norm, we get the same results as before. The mapping $\phi_w(x)$

is then defined as

$$\phi_w(x) = \text{argmax}_i\ x^T w_i. \tag{2.5}$$

From this point of view, both representations are equivalent. This special case is called *Dot-Product SOM*. If we imagine the input signal and the weight vectors as directed arrows, each input signal is mapped onto that neuron with the smallest angle between its weight vector and the input signal.

These mappings divide the input space into N_C devisions (where N_C is the number of output states)

$$R_s = \{x \in \mathcal{R} \mid \|x - w_s\| \leq \|x - w_i\| \ \forall i \in \mathcal{C}\}, \tag{2.6}$$

so that every input signal $x \in R_s$ activates the same neuron $s \in \mathcal{C}$. The weight vector w_s defines the center of the field. The polygon R_s is its *voronoi region*. The total of all polygons results in a partition of the input space, the *voronoi tessellation*. Each of these polygons is convex. An efficient algorithm to compute the voronoi tessellation for a given set of vectors is only known for the two-dimensional case (Preparata and Shamos [19]). The concept itself, however, is applicable to spaces of arbitrarily high dimensions.

Learning

The goal of the learning process is to reduce the average prediction error (distribution preserving) by simultaneous keeping the neighborhood relations (topology preserving). These two goals interact, and can't be reached together. Nevertheless, Kohonen's algorithm tries to find a compromise between them.

If a new sample x is presented to the net, *learning* is done by shifting the the *best matching weight* $w_s,\ s = \phi_w(x)$ of the *best matching neuron* s towards the sample x

$$\Delta w_s = \alpha\ (x - w_s), \qquad 0 \leq \alpha \leq 1. \tag{2.7}$$

To preserve topology, *topographically close* (in the output space) neurons are also modified.

$$\Delta w_i = \alpha\ h_{si}\ (x - w_s) \qquad i \in \mathcal{C}. \tag{2.8}$$

The neighborhood function h_{si} defines a neighborhood relation, that defines the *lateral interaction* between neuron s and i. This leads to a global and smooth ordering of the net, so that similar input vectors, are mapped to topological close neurons. α defines the *learning rate* and is usually a time decreasing monotonically function.

The learning equation should be defined according to the definition of the best matching neuron. E.g. the Dot-Product SOM uses a different update step:

$$w_s = h_{si} \left(\frac{w_i + \alpha x}{\|w_i + \alpha x\|} \right). \tag{2.9}$$

where h_{si} is set to 1 for each neuron that is a direct neighbor of s and to 0 for all other neurons. This learning step also ensures that the weight vectors keep normalized.

2.1.2 Algorithm

The algorithm suggested by Kohonen [12] can be split up into four basic steps. See the algorithm on the next page.

Kohonen's SOM algorithm

(Step 0: Initialization) Initialize the neighborhood relations to define the topology of the map. Initialize the weight vectors w_i with (random) values.

(Step 1: Selection) To train the net, an input vector x is chosen from the input data by random, according to the density distribution of the input data $P(x)$.

(Step 2: Response) The *best matching* weight vector w_s according to a distance measure (usually the Euclidean distance is used), is determined:

$$s = \text{argmin}_{i \in \mathcal{C}} \{\|x - w_i\|\} \tag{2.10}$$

(Step 3: Adaption) Modify the *weight vectors* of all neurons according to the learning process using a neighborhood relation in the net to create a smooth global ordering.

Step 0 is done once, when the net is set-up. The other steps 1 to 3 are repeated, until training is finished. This may be controlled by the number of iterations, or by a global error measure. Each step of this algoritm can be modified in several ways. To avoid redundancy, further sections will refer to these explanations if necessary.

Initialization

Initialization of the weight vectors can be done in various ways, depending on the knowledge of the input data [16].

- **Random Initialization:** The weight vectors w_c should approximate the input data. As this is often not known at the beginning of computation, a random initialization is chosen.

- **Using Initial Samples:** If initial information about the input data is available, it is profitable to choose the initial values of w_c corresponding to the density of the input data. This means, that the *weight vectors* w_c could be picked from the available input samples $x(t)$, preferably in a random order.

- **Linear Initialization:** Another approach is to use the two greatest eigenvectors of the autocorrelation matrix of x, that have the largest eigenvalues and then to span a (two-) dimensional linear subspace. A ordered array of vectors can now be used to initialize the net. As this net is already ordered, a faster convergence can be expected.

Response

The distance $d(x, y)$ used to find the best matching neuron, defines a metric on the input space and has to fulfill the following properties:

$$\begin{array}{lll} (ME\ 1) & d(x,y) \geq 0,\ d(x,y) = 0 \Leftrightarrow x = y & \forall x, y \in M \\ (ME\ 2) & d(x,y) = d(y,x) & \forall x, y \in M \\ (ME\ 3) & d(x,y) \leq d(x,z) + d(z,y) & \forall x, y, z \in M \end{array}$$

This ensures that no inhomogeneities occur that would tangle the learning process.

Adaption

- Update weights

  Learning is done, by updating the weight vectors of each neuron i, according to the *projection error* and their topological distance to the best matching neuron s. The projection error is defined as the distance between the input vector x and the weight vector w_i of the neuron i.

  $$w_i(t+1) = w_i(t) + \alpha(t) h_{si}(t)[x(t) - w_i(t)], \tag{2.11}$$

  where $t = 0, 1, 2, \ldots$ is the discrete-time coordinate and h_{si} is a *neighborhood function*, that defines how close a neuron s is to neuron i in the output space. $\alpha(t)$ is a time decreasing monotonically function, that defines the *learning rate*.

  Though the neighborhood function can be chosen arbitrary, two different choices for $h_{si}(t)$ occur most often: The definition of h_{si} as a *step-function*, and the definition as a Gaussian function. A step function is defined as:

$$h_{si} = \begin{cases} 1 & |s-i| \leq r_j \\ 0 & |s-i| > r_j \end{cases}, \tag{2.12}$$

where r_j defines the radius of neighborhood. If r_j is set to 1, no neighborhood relation is defined (*winner-takes-all*). A Gaussian function is defined by

$$h_{si} = \exp\left(-\frac{\|s-i\|^2}{2\sigma^2(t)}\right), \tag{2.13}$$

where $\sigma(t)$ is monotonically decreasing and defines the width of the kernel. The term $\|s-i\|$ may be replaced by any distance function.

If the neighborhood function is set according to the *mexican-hat function* where nearby neurons are activated, while distantly neurons are muted, the behavior of the net changes. This was analyzed by Xu, Krzyzak and Oja [18]. It shows, that each cluster is represented by a single neuron, as not-winning neurons are penalized and shifted apart. For that purpose, h_{si} was set to $+1$ for the winning neuron, -1 for its neighbors, and 0 for all other neurons. After a number of iterations, unused neurons are placed on the border, while each cluster is represented by a single neuron. The weights of these neurons are equal to the centers of the clusters.

Though Kohonen suggested a fixed *learning rate* first, $\alpha(t)$ should be set according to

$$\alpha(t) = \alpha_{\text{start}} \left(\frac{\alpha_{\text{end}}}{\alpha_{\text{start}}}\right)^{t/t_{\max}}, \tag{2.14}$$

to ensure fast convergence. Another possible choice for $\alpha(t)$ is the harmonic series $1/t$. See [16] for a detailed analysis.

The selection of the sphere of influence from h_{si} is very important for the behavior of the net. If it is to small at the beginning, the net will stay in an non-optimal state, and may not become completely "unfolded". Otherwise, if the neighborhood size stays to large, the net will shift the neurons in each iteration towards the new sample and the net will not converge. If the neighborhood size becomes zero, the SOM algorithm converges as proved by Yin and Allision [20]. To guarantee convergence, two additional conditions have to be fulfilled:

$$\sum_{t=0}^{\infty} \alpha(t) = \infty, \quad \sum_{t=0}^{\infty} \alpha(t)^2 < \infty \tag{2.15}$$

2.1.3 Analysis

The benefit of Kohonen's SOM algorithm is its ability to fit the net into different input data spaces, approximating the density distribution of the data $P(x)$ in some way. An important property is the proof of convergence under certain conditions. Also non-trivial mappings are found, without any a-priori knowledge of the input-distribution.

As a drawback, one has to know the expected dimension of the input data, because the dimension and size of the net is defined at the beginning. The dimension has also to be homogeneous over the whole input space. This is especially a problem, if the input data, breaks up into clusters, or has areas of different density.

It is important to make a difference between the dimension of the input data, and the dimension of the input space. The dimension of the input space has no influence at all, as the net tries to approximate the input manifold. If the input space is very high dimensional but the input data is not, the map can be used to eliminate the unnecessary dimensions. But if the dimension of the net is lower than the dimension of the input data, the net may corrugate. E.g. using a one dimensional net, to represent a even distributed two-dimensional input data, results in a net, that is comparable to a peano-curve (see figure 2.3). This becomes clear, if one sees Kohonen's algorithm as a process of minimization of the the average distance between the sample points and their nearest reference-vector. For this minimization problem a peano curve is the optimal solution.

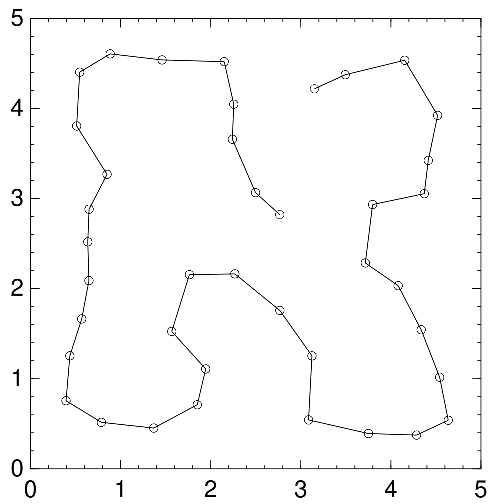

Figure 2.3: A one dimensional net, learning a two dimensional square

This may become a problem, as neighborhood relations are not represented in a correct way. Nearby neurons in the input space can be far apart in the output space. To avoid this, the right dimension for the net has to be chosen. In contrary if the input data would lay on a diagonal of the square, the net would map the neighborhood relations in a correct manner, because the input data distribution has the same dimension as the net. See 2.8 for how to decide the quality of the mapping.

2.2 Neural Gas plus Competitive Hebbian Learning

The *Neural Gas* developed by Martinetz and Schulten [22, 24], expands the *k-means* algorithm, by adding a neighborhood relation. This allows not only to map the input data onto k nodes, but also to conceive neighborhood relations of the input data.

For each neuron the distance to the given sample is computed. The two nearest neurons are selected and a connection between them is created. This is called *Competitive Hebbian Learning.* Hebbian means, that activated connections are strengthened, while unused ones are clammed. It is competitive, because this process is repeated and inactive connections are removed in time.

As opposed to Kohonen's model the neighborhood function is not defined through the map, but as similarity of the *weight vectors.* This is necessary as the net is unconnected at the beginning. As no topological order is presumed, it is very flexible and creates the topology of the net on its own.

2.2.1 Algorithm

The algorithm differs only in the *response* and the *adaption* step from Kohonen's model, so only these differences will be discussed in detail.

We denote the set of connections (edges) with $\mathcal{E}, \mathcal{E} \subset \mathcal{C} \times \mathcal{C}$. At the beginning, no connections between the neurons are defined,

$$\mathcal{E} = \emptyset. \tag{2.16}$$

For a unit c we denote the set of its direct topological neighbors with $\mathcal{N}_c$:

$$\mathcal{N}_c = \{i \in \mathcal{C} \,|\, (c, i) \in \mathcal{E}\}. \tag{2.17}$$

Response

First the *best matching* weight vector w_s is determined as is done in the original algorithm. For all other weight vectors w_i, the number of weight vectors k_i, that are closer to the given sample is computed. This measure is used as a distance between two neurons, which is defined on the input space. It replaces the distance function in the output space, used in the neighborhood function by Kohonen.

Adaption

- Update Weights

Adaption is done in two steps. First all weight vectors are updated. Let $\alpha(t)$ be the *learning rate*, and λ be a measure of how many vectors should be modified. Then the neighborhood function h_{si} is defined as

$$h_{si} = \alpha(t)e^{-\frac{k_i}{\lambda(t)}}, \tag{2.18}$$

where λ is defined as a time-decreasing function,

$$\lambda(t) = \lambda_{\text{start}} \left(\frac{\lambda_{\text{end}}}{\lambda_{\text{start}}} \right)^{t/t_{\max}} \tag{2.19}$$

- Update net structure

  In a second step *Competitive Hebbian Learning* is used to create a new connection, between the best s_1, and the second best s_2 matching weight vector.

  $$\mathcal{E} = \mathcal{E} \cup \{(s_1, s_2)\} \tag{2.20}$$

  Connections are marked with a time stamp, and are removed, if they are not updated frequently. The time stamp of a new connection is set to zero. All other time stamps are increased. If a connection is older than a predefined maximum age L, it is removed.

2.2.2 Analysis

The benefit of this model, is it's ability to create a topological structure on its own that fits to the input space. If the input data divides in several distinct clusters, the net will also split up in several parts. The number of connections can be controlled by courtesy of the parameters.

On the other hand, this can results in a very inhomogeneous structure, with no fix network dimension. This may be a disadvantage when displaying the results, or for analytical reasons.

The computation of k_i is very time-consuming, as for each new sample, the list of neighbored neurons has to be sorted. Compared to the original algorithm, computation time is very high.

2.3 Growing Neural Gas

This model, developed by Fritzke in 1995 [25], uses a flexible number of neurons. These neurons are connected, using *Competitive Hebbian Learning* as in the Neural Gas. A local error measure is used, to determine where to insert new neurons. In contrary to the Neural Gas model, a neighborhood relation comparable to Kohonen's model is used to train the net.

2.3.1 Algorithm

The algorithm is an extension of the *neural gas* algorithm, so we will only discuss the additional parts here.

First the set of neurons $\mathcal{C}$ is initialized with two unconnected neurons

$$\mathcal{C} = \{c_1, c_2\} \tag{2.21}$$

Their weight vectors are set according to $P(\xi)$, the distribution of the input data. The set of all connections $\mathcal{E}$ is initialized to the empty set

$$\mathcal{E} = \emptyset. \tag{2.22}$$

The local error variables $E_i, i \in \mathcal{C}$ are set to zero, for all neurons.

Adaption

Adaption includes a number of different steps. First a new connection between the best and the second best matching neuron is created as is done by the Neural Gas. Then the weights of the neurons are modified to improve the quality of the net. Finally it is decided if and where to add new neurons.

- Update net structure I

  A connection between the best and the second best matching weight vector is created as is done in the *Neural Gas* algorithm

  $$\mathcal{E} = \mathcal{E} \cup \{(s_1, s_2)\}, \tag{2.23}$$

  and the time stamp is set to zero.

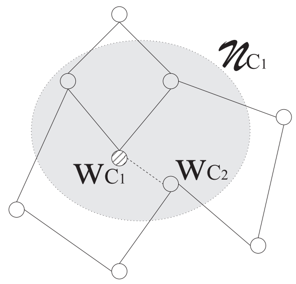

Figure 2.4: Growing Neural Net, Adding a new connection

- Update Weights

  – The squared distance between the input data and the winning weight vector is computed and stored in the local error variable:

  $$E_{s_1}(t+1) = E_{s_1}(t) + \|x - w_{s_1}\|^2. \tag{2.24}$$

- Update the best matching weight vector s_1 and its direct topological neighbors $\mathcal{N}_{s_1}$ using fix learning rates α_b and α_n:

$$\begin{aligned} w_{c_1}(t+1) &= w_{s_1}(t) + \alpha_b(x - w_{s_1}) \\ w_i(t+1) &= w_i(t) + \alpha_n(x - w_i) \qquad \forall i \in \mathcal{N}_{s_1}. \end{aligned} \tag{2.25}$$

- Increment the time stamp of all connections, and remove those, that exceed the maximum age L. If a neuron has no connections left, remove it as well.

- Update net structure II

  If the number of input signals generated so far is an integer multiple of a predefined parameter, insert a new neuron as follows:

  - Determine the neuron q with the maximum accumulated error:

$$q = \text{argmax}_{c \in \mathcal{C}} E_c \tag{2.26}$$

  - Determine the direct neighbor f of q with the maximum accumulated error:

$$f = \text{argmax}_{c \in \mathcal{N}_q} E_c \tag{2.27}$$

  - Add a new neuron r and define its reference vector between the two neighbors:

$$\mathcal{C} = \mathcal{C} \cup \{r\}, \qquad w_r = (w_q + w_f)/2. \tag{2.28}$$

  - Connect this new neuron r to q and f, and remove the old connection between them:

$$\mathcal{E} = \mathcal{E} \cup \{(r,q),(r,f)\} \ / \ \{(q,f)\}. \tag{2.29}$$

  - Modify the error variables of q, f and r:

$$E_q(t+1) = E_q(t) - \epsilon E_q(t) \tag{2.30}$$
$$E_f(t+1) = E_f(t) - \epsilon E_f(t) \tag{2.31}$$
$$E_r(t+1) = (E_q(t+1) + E_f(t+1))/2. \tag{2.32}$$

- Update error variables

  Decrease all error variables by a fix amount:

$$E_c(t+1) = E_c(t) - \beta E_c(t) \qquad \forall c \in \mathcal{C}. \tag{2.33}$$

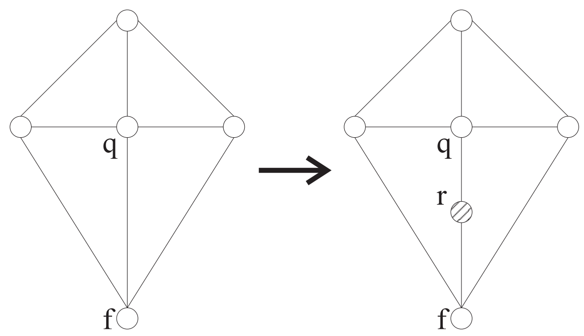

Figure 2.5: Growing Neural Net, Adding a new neuron

2.3.2 Analysis

This model is able to create a network structure, without any assumptions about the dimension or size of the network. Neurons are always inserted at that position, where they are needed most, so that the net needs only very few redistribution. This accelerates computation and ensures, that the net is always in a good condition. As opposed to the Neural Gas algorithm, where the list of neighbors had to be computed in each training step, only the two best matching nodes are required. This reduces computation time.

The final net has no fix network dimension. This leads to the same problems, mentioned at the analysis of the *Neural Gas*.

2.4 Growing Cell Structures

This model developed by Fritzke 1993 [23], is closely related to the *growing neural gas* model, but to overcome the problems mentioned before, it uses a fixed network dimension. A fix network dimension is useful if one wants to display the results or use them in further computations. Growing is done by not only adding a new connection, but a complete simplex to the net at the point with the maximal local error measure. For the two dimensional case this are triangles, for the higher dimensional case (hyper-) tetrahedrons.

2.4.1 Algorithm

The algorithm is basically the same, as the one used for the *growing neural gas*, except that insertion of a new connection is different and that the the first Update Net step is omitted (including that part, removing old edges).

Adaption

- Update net structure

If the number of input signals generated so far is an integer multiple of a predefined parameter, insert a new neuron as follows:

– Determine the neuron q with the maximum accumulated error:

$$q = \text{argmax}_{c \in \mathcal{C}} E_c \tag{2.34}$$

– If the longest edge going out from q is (q, f),

$$f = \text{argmax}_{i \in \mathcal{N}_q} \|w_q - w_i\| \tag{2.35}$$

then split this connection and insert a new neuron r.

$$\mathcal{C} = \mathcal{C} \cup \{r\}, \qquad w_r = (w_q + w_f)/2. \tag{2.36}$$

– Connect this new neuron r to q, f and all common simplices (that are all simplices, which contain the edge (q, f). Remove the old connection (q, f).

– Modify the error variables of r and all its neighbors:

$$E_i(t+1) = E_i(t) - \frac{\epsilon}{|N_r|} E_i(t) \qquad \forall i \in \mathcal{N}_r. \tag{2.37}$$

$$E_r(t+1) = \frac{1}{|N_r|} \sum_{i \in N_r} E_i. \tag{2.38}$$

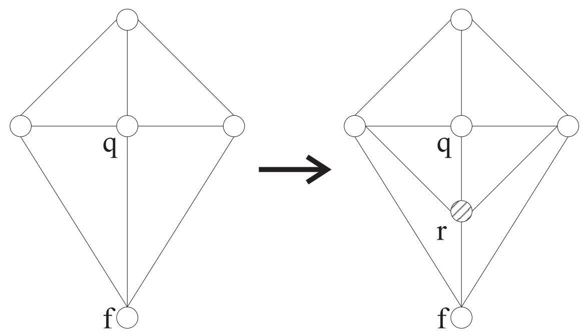

Figure 2.6: Growing Cell Structure, Adding a new neuron

To allow the network to split up in different parts, deletion of neurons is possible. For that purpose neurons with very less samples are determined and removed.

- Removal of cells

  Cells laying in regions with very few input samples, can be removed to enable the net to cover other regions more efficiently. As the overall density of the input data $P(x)$ is unknown, the relative signal frequency of a cell to the size of its Voronoi field might be used. To get a local estimate for the probability density near w_c we compute

$$\tilde{p_c} = h_c/|F_c|, \tag{2.39}$$

where h_c is the relative signal frequency of c, and F_c denotes the Voronoi field of c. Neurons with a relative density $\tilde{p_c} < \eta$ (η is a predefined limit) are removed periodically.

As the computation of $|F_c|$ might be very complex, one can approximate the size of the Voronoi field by a rectangular hypercube

$$\tilde{f_c} = (\bar{l_c})^n, \tag{2.40}$$

where $\bar{l_c}$ is computed according to

$$\bar{l_c} = \frac{1}{\#\mathcal{N}_c} \sum_{i \in \mathcal{N}_c} \|w_c - w_i\|. \tag{2.41}$$

$\#\{\cdot\}$ denotes the cardinality of a set. As great Voronoi regions with a small but very dense center, might be undervalued, the approximated density should be normalized over all hypercubes:

$$\hat{p} = \tilde{p} \sum_{c \in \mathcal{C}} \tilde{f_c}. \tag{2.42}$$

If the distribution of the input data is very low dimensional in comparison to the input space, a lower dimensioned hypercube might be used.

2.4.2 Analysis

This algorithm results in a good fitting net, with flexible size but predefined dimension. As simplices are used instead of a grid-structure, visualization may be difficult if not projected to the input space. Fritzke suggested a method for this problem in 1993 [23].

As this model uses a local error-measure to determine the position of where to insert the next new simplex, other goals then distribution-preserving may be achieved. The average quantization-error may result in a good vector-quantification. Also supervised variations are possible. Fritzke [23] used a combination of *radial basis functions* (RBF) and the growing cell structure and achieved good results for clustering and classification. The benefit of his approach is, that the clusters lead to discrete structures that are easy to process.

2.5 Growing Grid

Developed in 1995 by Fritzke [26], this model is a combination of ideas from the growing-cells using a classic grid, as underlying structure. Growing is done, by inserting a whole row or column at the point with the highest local error measure.

2.5.1 Algorithm

As the grid is growing very fast, the algorithm is divided into two parts. The *growing phase* and the *fine tuning phase.*

Starting with a 2^n grid, growing is done following the same idea, used in the *growing neural gas*. After determining the *best matching weight vector* w_s, an adaption step is done.

Adaption

- Update Weights I

  Using the L_1 norm (witch is also called *city block metric*), the weight vectors according to the neighbors of c are shifted towards x using a Gaussian neighborhood relation (see equation 2.11 and 2.13). $\alpha(t)$ is set to a constant value α_0 while growing. Then the local error variable E_s is increased by 1.

  This learning process is repeated, until all neurons were updated λ_g times in average. For a $k \times m$ network, that would be after $k \times m \times \lambda_g$ steps.

- Update net structure

  If enough learning steps are done, the neuron with the maximum error value is determined:

  $$q = \text{argmax}_{c \in \mathcal{C}} E_c. \tag{2.43}$$

  This is the neuron, that represents the most input samples. Now the neighborhood neuron, with the most different reverence vector is determined

  $$f = \text{argmax}_{c \in \mathcal{N}_q} \|w_c - w_q\|. \tag{2.44}$$

  Without loss of generality we assume, that q and f both belong to the i-th row (otherwise they would belong to the same column, witch can be treated accordingly). Furthermore we assume, that $q = s_{ij}$ and $f = s_{ij+1}$.

  A new column j' (with k neurons) is inserted between the columns j and $j+1$. The weight vectors of this new column are interpolated from its neighbors:

  $$w_{rj'} = (w_{rj} + w_{rj+1})/2 \qquad 1 \leq r \leq k. \tag{2.45}$$

Then all local error variables are reset to zero.

While a *stopping criteria* is not fulfilled (for example the final number of neurons is not reached), a new learning step is done.

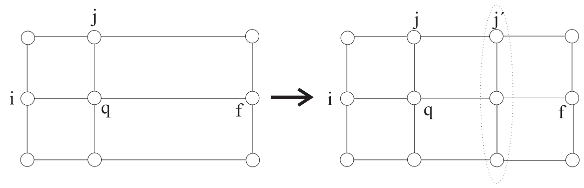

Figure 2.7: Growing Grid, Adding a new row

- Update Weights II

  If the net contains enough neurons, so that no more growing is necessary, a few fine-tuning steps can be performed. Fritzke suggested $t'_{\max} = k \times m \times \lambda_f$ learning steps, with a learning rate

$$\alpha(t') = \alpha_0(\alpha_{\max}/\alpha_0)^{t'/t'_{\max}}, \tag{2.46}$$

  where λ_f is the number of learning steps, that should be performed for each neuron in average. This step is an application of Kohonen's original algorithm to the complete grown net.

2.5.2 Analysis

The growing grid model is a compromise between flexibility and predefined structure. It achieves great results, if the input data is distributed "rectangular" in some way. As the size of the net can be defined using a flexible stopping criteria, it is possible to adjust the quality of the net very easy. On the other hand, if the density of the input data varies very much over the input space, the grid may be a to strong structure, to represent these inhomogeneities.

2.6 Growing Hypercubial Output Space (GSOM)

The GSOM Model, developed by H.U. Bauer and Th. Villmann in 1995 [28] is in its basic idea also a growing grid algorithm. As opposed to Fritzke, the GSOM can also grow into a new dimension, so that the network dimensionality has not to be known a priori.

2.6.1 Mathematical model

Bauer and Villmann suggested to use the average projection error, to decide where to add a new row of neurons or a new dimension. For this purpose, the Voronoi-region of a neurons r weight-vector w_r is decomposed into the different directions, induced by its topological neighbors to define a new basis $b_1 \ldots b_d$.

Suppose, r has the grid-coordinates $(r_i, r_j, r_k, 1, \ldots)$, with network dimensionality $d = 3$ (the coordinates r_i for $l > d$ are set to 1 per definition. Then r has two neighbors in each dimension. The direction-vector for the first-coordinate can be computed as the normalized difference of the reference vectors of these two neighbors:

$$b_1(r) = \frac{w_{(r_{i+1}, r_j, r_k, 1, \ldots)} - w_{(r_{i-1}, r_j, r_k, 1, \ldots)}}{\| w_{(r_{i+1}, r_j, r_k, 1, \ldots)} - w_{(r_{i-1}, r_j, r_k, 1, \ldots)} \|}, \tag{2.47}$$

for the other two dimensions accordingly.

In general, if a input signal x is mapped to the neuron r, the quantization error $\Delta w_r(x) = x - w_r$ can now be written as

$$\begin{aligned} \Delta w_r(x) &= a_1(x)b_1(r) + a_2(x)b_2(r) + \ldots + x' \\ &= \sum_{i=1}^{d} a_i(x)b_i(r) + x'. \end{aligned} \tag{2.48}$$

Where a_i is the amplitude of the i-th basis vector and x' the remaining error.

The idea is, to insert a new row in that dimension, with the largest average error amplitude a_i. To suggest, whether to add a new dimension or not, a principle component analysis is done. Let e_{PGA} be the direction of largest variability, then the remaining error x' can be written as

$$x' = a_{d+1}(x)e_{PGA} + x''. \tag{2.49}$$

For each input signal, this amplitude is normalized and then averaged over all signals

$$\tilde{a}_i = \sum_x \frac{|a_i(x)|}{\sqrt{\sum_{j=1}^{d+1} a_j^2(x)}}, \qquad i = 1, \ldots, d, d+1 \tag{2.50}$$

As the deviation into a direction will decrease by adding a new row, this factor is also taken into account as its root.

The growing direction i is selected according to

$$i = \text{argmax}_j \left(\sqrt{\frac{n_j}{n_j + 1}} \tilde{a}_j \right), \qquad j = 1, \ldots, d, d+1 \tag{2.51}$$

As no local error measure is available, new rows are inserted in the middle of the grid. In case of a new dimension, the net should grow into the direction of the respective principal components.

To enable the net to redistribute, the learning rate $\alpha(t)$ should be increased after each growing step.

2.6.2 Algorithm

Adaption

Adaption is done in the same way, as suggested by Kohonen. After a fix number of learning steps, it is checked, whether new rows or dimensions have to be added. It would also be possible, to split the algorithm in the same two parts as in the *growing grid* algorithm.

The main difference is, that no local error measure was used and the learning rate $\alpha(t)$ has to be increased after each growing step.

2.6.3 Analysis

As no a priori knowledge about the dimension or size of the grid has to be known, this model allows very easy ad hoc processing of high dimensional data. To achieve this goal, a lot of computation is necessary: For each neuron, a principle component analysis over all sample points in its voronoi region has to be done and insertion is always done in the middle, so that the whole net has to redistribute.

This results in a very complex and slow algorithm, so that the added flexibility is quite costly. Fritzkes algorithms deliver comparable results, and allows a real incremental computation, operating only with local measures and modifications.

2.7 Common SOM objectives

As flexible as competitive learning methods are, as different are the goals, that may be aspired by this methods. Depending on the point of view, one may try to minimize the average error of prediction, or on the other hand, maximize the entropy, so that every vector has the same chance to be chosen.

It is clear, that these goals can not be reached simultaneously in many cases. So it is an important part of the design process, to clear the objectives of the system.

Error Minimization

An important goal in many cases is the minimization of the *expected quantization error*. This means, that the weight vectors are homogeneous distributed over the input data, so that the distance of an input signal, to the nearest weight vector is minimized on an average.

Using Voronoi regions, the *expected quantization error* can easily be computed:

$$\mathrm{E}(p(x), \mathcal{A}) = \sum_{c \in \mathcal{A}} \int_{V_c} \|x - w_c\|^2 p(x) \, \mathrm{d}x \tag{2.52}$$

Where $p(x)$ is the input signal distribution and $w_c, c \in \mathcal{A}$ is the weight vector corresponding to c. A Voronoi region $V_c, c \in \mathcal{A}$ contains all points, that are closer to c, than to any other point of $\mathcal{A}$.

Minimizing the average error is very important in *vector quantization* (Linde et al., 1980; Gray, 1984). The information loss, caused by quantization, is equal to the distance to the nearest weight vector (called *codebook vector*).

Entropy Maximization

If every weight vector should have the same chance to be chosen for a randomly selected input signal x, every voronoi region has to contain the same number of input data points.

$$P(s(x) = c) = \frac{1}{\|\mathcal{A}\|} \; (\forall c \in \mathcal{A}) \tag{2.53}$$

The advantage of this approach is the robustness of the resulting system, because the influence of one vector is limited fraction of the data.

Feature Mapping

Some networks are able, to reduce high-dimensional input data, to a lower dimensional hyper-plain in a *topology preserving* manner. This has been denoted *feature mapping* and is also very useful in data visualization.

The quality of this projection can be determined using a variety of measures. See section 2.8.1 and 2.8.2 for more information.

Clustering

In a few cases, it is possible to cluster the input data, that means, that similar data is grouped together. For good results, the intra-cluster variance should be small, and the inter-cluster variance large. In supervised environments, the quality of prediction may be used as a quality measure.

2.8 Valuation measures

To check, whether the topology of the input space is mapped in a correct manner to the map, two functions have been suggested. The *topographic product* introduced by Bauer and Pawelzik in 1992, and the *topographic function* from Villmann et al. (1994). Both try to compare the ordering of neighbors, instead of absolute distances.

2.8.1 Topographic product

This method compares the neighborhood sequence for all neurons between the input and the output space. For this purpose, two sequence functions $n_k^{\mathcal{C}}(s)$ and $n_k^{\mathcal{R}}(s)$ are defined, where $n_k^{\mathcal{C}}(s)$ denotes the k-th neighbor of s, with distances measured in the output space. $n_k^{\mathcal{R}}(s)$ denotes the k-th neighbor of s, with distances evaluated in the input space between w_s and $w_{n_k^{\mathcal{C}}}(s)$.

First the quantity

$$P_3(s,k) = \left(\prod_{l=1}^{k} \frac{d^{\mathcal{R}}\left(w_s, w_{n_k^{\mathcal{C}}}(s)\right)}{d^{\mathcal{R}}\left(w_s, w_{n_k^{\mathcal{R}}}(s)\right)} \cdot \frac{d^{\mathcal{C}}\left(s, n_k^{\mathcal{C}}(s)\right)}{d^{\mathcal{C}}\left(s, n_k^{\mathcal{R}}(s)\right)} \right)^{\frac{1}{2k}} \tag{2.54}$$

is defined. The topographic product is finally computed as an average over neighborhood orders k and all neurons s

$$P = \frac{1}{N_C(N_C-1)} \sum_{s \in \mathcal{C}} \sum_{k=1}^{N_C-1} log(P_3(s,k)). \tag{2.55}$$

Where N_C denotes the total number of neurons in the output space.

The value of P can be positive or negative. A value nearby 0 indicates a good topological approximation of the input data, a too high or too low value indicates some violated neighborhood relations.

2.8.2 Topographic function

The topographic function is based on a stringent mathematical foundation. It uses the input data to evaluate the quality of the approximation instead of a neighborhood ordering. For details see ..

The basic idea is, to compare two neighborhood relations. A strong neighborhood relation is defined in the input space, using the Delaunay-graph $\mathcal{D}_R$. A Delaunay-graph is defined, by connecting all neurons, whose Voronoi-regions share an edge. Using this graph, the distance $d^{\mathcal{D}_V}(w_s, w_{s'})$ is defined as the length of the minimal path on the Delaunay-graph between s and s'. In the output space, a weak neighborhood, based on the maximum norm $\|s - s'\|_{max} = max_{k=1}^{d}|s_k - s'_k|$ is defined. A map is called neighborhood preserving, if neurons $w_r, w_{r'}$, connected by the Delaunay-graph, are mapped onto weakly neighbored neurons r and r'. In the opposite if two neurons r, r' are neighbored, their weight vectors $w_r, w_{r'}$ should be connected by $\mathcal{D}_V$. We define for each neuron $s \in \mathcal{C}$

$$\begin{aligned} f_s(k) &= \#\{s' \mid \|s - s'\|_{max} > k \, , \, d^{\mathcal{D}_V}(w_s, w'_s) = 1\}, \\ f_s(-k) &= \#\{s' \mid \|s - s'\|_{max} = 1 \, , \, d^{\mathcal{D}_V}(w_s, w'_s) > k\}. \end{aligned} \tag{2.56}$$

with $k = 1, \ldots, N_C - 1$. $\#\{\cdot\}$ denotes the cardinality of a set. The topographic function $\phi_{\mathcal{C}}^{V}$ then is defined by

$$\phi_{\mathcal{C}}^{V}(k) = \begin{cases} \frac{1}{N}\sum_{r' \in \mathcal{C}} f_{r'}(k) & k > 0 \\ \phi_{\mathcal{C}}^{V}(1) + \phi_{\mathcal{C}}^{V}(-1) & k = 0 \\ \frac{1}{N}\sum_{r' \in \mathcal{C}} f_{r'}(k) & k > 0 \end{cases} \tag{2.57}$$

Chapter 3

Component analysis

In this chapter a generalized framework for SOM algorithms will be developed. In contrast to most implementations where each algorithm is presented in an own part, I will try to create a flexible framework that allows to reuse not only the interface of the algorithm (like input- / output handling), but also main parts of the algorithms themself. For that purpose Kohonen's algorithm is analyzed and divided in its basic components. This allows to exchange these parts independently and to create new variants of the algorithm very easy. Then these components are generalized and analyzed with respect to the growing algorithms described in the previous chapter. It is shown, that most methods are common to all algorithms, and that only a small number of methods has to be redesigned to create a new algorithm.

3.1 Generalization

If we take a look at the basic algorithm in section 2.1.2, we can obtain the four basic steps:

Initialization of the maps structure and the weights.

Selection of a new input sample.

Response of the best matching neuron.

Adaption of the nodes according to their distance to the best matching node.

While *Initialization* is only done once at the beginning, the other three steps are repeated until some *break condition* is fulfilled. Therefor these steps will be sticked together to a general TrainNet step to allow easier reuse and modification. In addition to the basic algorithm, growing nets need a special *net adaption* step, that is performed, if a given test is satisfied.

If we take this into account and rename the steps a bit, we get a more detailed concept:

```
Initialization()

TrainNet():
   while (not BreakCondition())
      GetNewSample()
      BestMatching()
      Adaption()
      if (AdaptionNetTest())
         AdaptionNet()
   end
end
```

Each command ending with round brackets is a generalized function, and will be discussed in the following sections in detail.

- Initialization

  Initialization can be split up in two parts: Generation of the maps internal structure and initialization of the neurons weights. This is done, to separate the part concerning the networks structure, which is depending on the map used (for example a grid) and the part, dealing with the data stored in the map. This allows greater reusability of each part, as the initialization of the maps weights is (widely) independent of the maps structure.

  ```
  Generate() map structure
  Initialize() node weights
  ```

  Generation depends a lot on the used learning method. For example a fix network structure like a $n \times n$ grid (Kohonen, GrowingGrid, GSOM) or just a number of initial nodes may be used ((Growing-) Neural Gas).

  Initialization depends on the knowledge of the input data (see section 2.1.2) and sometimes of the network structure. This part is algorithm independent, but has to be handled in a flexible way too. This enables the user to test, how a algorithm can deal with different initial setups (random or pre-ordered).

- TrainNet

  This step offers a common interface between the map and the algorithm. It covers all steps, necessary to train the net.

```
while (not BreakCondition())
   GetNewSample()
   BestMatching()
   Adaption()
   if (AdaptionNetTest())
      AdaptionNet()
end
```

- BreakCondition

  The BreakCondition may be any kind of test, whether the goals of the learning process have been achieved. In general only a fix amount of iterations is computed, so that the BreakCondition is a simple test, whether $t < t_{max}$. Other possibilities are to test, if a error measure is below a given bound or, for a growing algorithm, if enough nodes have been generated so far. It is very important to combine any test with a check on the maximum number of iterations, to prevent the algorithm from running forever in unpredictable cases.

- GetNewSample

  New input samples may come from various sources. For simple tests, a randomly generated sample may be sufficient, but most times input samples will be read in from a given dataset. To be free from limitations of the interface, it is very important that any kind of input data can be read in convenient.

- BestMatching

  Kohonen used the Euclidean norm to find the node, whose weight has minimal distance to the given sample. From a general point of view, any distance measure can be used.

  The *Distance Measure* depends on the input data (and further on the weights, which are of the same type). For example, if undirected graphs are read in, the distance between two graphs may be the number of nodes with a different number of edges.

```
for all nodes
   curDist = Weight.Distance(curNode, sample)
   if (curDist < bestDist)
      bestNode = curNode; bestDist = curDist
   end
end
```

- Adaption

  Adaption takes each nodes, computes the distance to the winning node and the learning rate to adapt the nodes weight accordingly. This means, that nearby nodes are shifted towards the sample, while far away nodes are not modified at all or shifted apart. This computation is done for each node and therefor covered by a separate *UpdateNode* method. Eventually edges and nodes may be added or removed in an optional *UpdateNet* step. See section 3.2.2 and 3.2.3 for a detailed analysis.

```
for all nodes
   UpdateNode()
end
UpdateNet()
```

- AdaptionNetTest

  If the AdaptionNetTest criteria is fulfilled, an AdaptionNet step is performed. Usually this criteria is a test, whether the number of iterations is a multiple of a fix parameter ($t \ mod \ t_{AdaptNet} = 0$).

```
if (t mod t_net = 0)
   AdaptionNet
```

- AdaptionNet

  The adaption of the net structure (see section 3.2.6) depends on the algorithm used. The common structure is, to locate that part of the map, with the highest demand (in what sense ever) for a new node and then to insert a new node (or row, or whatever). The demand to insert additional nodes can be measured by a local error measure, that is computed while learning or by a analysis of the current map state.

3.2 Component classes

Now the different components will be discussed in detail. Each one is compared for all algorithms to find similarities and distinctions.

3.2.1 Map data

As the algorithm uses different map structures (gas, grid etc.) a dynamic structure is necessary to enable a wide range of possible setups. For the basic algorithm only an initial setup and a distance function is necessary. For growing algorithms the basic modifications (add, remove nodes / edges) should be possible too. Additional features like adding a vertices or a row are necessary for certain algorithms. All these operations need to be possible on the general map structure.

The best way to represent such a dynamic structure is to use an undirected graph. This allows complete freedom for the choice of dimension and the number of nodes and edges. With a graph as underlying structure it is also very easy to find the neighbors of a node, without information of the general structure, or to define a distance between two nodes (minimal path). To store the weight and other local information, it is necessary that to each node additional data can be associated.

The additional needed methods for each algorithm are described in the following list:

Neural Gas

A free network with a fix number of nodes is needed. Edges are created and removed on demand. Here a (not necessarily connected) graph is used.

Growing Neural Gas

As in the neural gas, no fix network structure is given. Additionally new nodes need to be inserted. Also a check, whether a node has any connections left is needed (resp. has to be possible).

Growing Cells

A network with a fix dimension consisting of n-dimensional simplices is used. For the 2-dimensional case these are triangles, for the 3-dimensional one tetrahedrons and so on. When adding a new node, not only an edge is split, but all common simplices. This ensures, that the network dimension stays constant. Removing of a node, has to be handled accordingly.

Growing Grid

Uses a fix dimensioned grid, with varying size. Here not only nodes and edges, but complete rows are inserted at arbitrary positions.

GSOM

In opposite to the Neural Gas, the network dimension is not fixed. Therefor it is also needed, to add a new dimension by duplicating the grid, and connecting each node with its copy (imagine, how to create a cube from a square).

3.2.2 Adaption

The Adaption step is the kernel of any algorithm. It modifies the nodes weights and (depending on the algorithm used) creates and removes nodes and edges. As growing is not done every learning step, it is useful to split the adaption in two parts like the Initialization step. One, that adapts the weights, and one that modifies the net structure. The "weight update step" updates each node of the map. Eventually some preprocessing may be done before to fasten computation. Some algorithm also modify the net in each learning step (mostly only edges are created/removed). For that purpose, a separate "update net step" is needed. As this step is done every iteration and also closely related to the adaption step used, no extra component will be created, but it was treated as a part of the adaption step.

UpdateNet

Updating the net at this stage, is only done in the Neural Gas and the Growing Neural Gas algorithms. This is necessary, because both use no predefined network structure. Instead edges are created and removed on demand. Because this sub step is very specific, no separate component class was designed.

Neural Gas

> A new edge between the best and the second best matching node is created and the edge-age is set to zero. For all other nodes the age is increased and if an edge is older than a given boundary, it is removed.

Growing Neural Gas

> The handling of the edge-ages is similar. Additionally nodes, that have no edges remaining are removed as well.

3.2.3 UpdateNode

Kohonen updated the nodes, with respect to two measures. The Learning Rate and a Neighborhood Function. Both may vary depending on the algorithm. Though Kohonen used the Learning Rate inside the Neighborhood Function, it is simpler to separate this two functions and multiply them afterwards. Sometimes it may also be necessary to do some additional computations. Especially the *Neural Gas* uses a completely different approach.

Learning Rate (see section 3.2.4) and Neighborhood Function (section 3.2.5) are very important functions, as they determine the behavior of the learning process. They may vary between different problems and algorithms. By changing one or both, the learning process can be controlled. One goal of the framework is to allow easy exchange of these components to compare the results. As these two components are similar in all variants of Kohonen's algorithm, they can be reused very easy.

```
alpha = LearningRate()
hsi   = NeighborhoodFunction()
Node.Weight = Node.Weight +
              alpha * hsi * (sample - Node.Weight)
```

Neural Gas

Before the weights can be updated, it is useful to compute the number of nodes closer to the sample for each node. Then the node weights can be modified using the according Neighborhood Function.

Growing Neural Gas

The nodes are updated using the original method. For the winning node, the prediction error is summed up. This is the distance, between the nodes weight, and the sample. This enables the algorithm to find nodes, that represent a large number of widely spread nodes.

Growing Cells and Growing Grid

Updating the nodes and the local error variables is done as is the Growing Neural Gas.

GSOM

Here only a simple update of the nodes is done as known from Kohonen. No special computation is necessary.

3.2.4 Learning Rate

The learning rate (α) can be modified to define the strength of the learning process. However it is mostly independent from the learning method used (see equation 2.14). for examples).
Possible choices for the learning rate are:

Fix learning rate

The learning rate is set to a fix value, that is relatively small. This was first introduced by Kohonen [12]. The net is able to follow also non stationary data distributions, as learning doesn't increase with time. The disadvantage is, that the net does not converge.

```
alpha = alpha_fix
```

k-means

The learning rate is set according the harmonic series $\alpha = 1/t$. Each node represents the exact arithmetic mean of the points, it was winner before. The net, however, is also not guaranteed to be convergent.

```
alpha = 1 / t
```

Exponentially Decaying

If the learning rate is defined as in equation 2.14 with accurate parameters, convergence is ensured. The learning rate is higher at the beginning, to enable the net to arrange the nodes over the input space (and "unfold" the net if necessary). At the end, the learning rate decreases slower, to allow only small redistributions.

```
alpha = alpha_start *
        (alpha_end / alpha_start) ^ (t / t_max)
```

While the above methods may be used for all algorithms, the GSOM algorithm needs an additional modification:

GSOM

As the GSOM algorithm inserts new rows always in the middle of the grid, it is necessary that all other rows can redistribute. Therefore the learning rate is increased after each net adaption step.

3.2.5 Neighborhood Function

The neighborhood function (h_{si}) defines, how much a given node should be updated, according to his distance to the best matching node (see equations 2.12 and 2.13). Nevertheless some methods (in their original form) use special variations of the neighborhood function. It should be mentioned, that nevertheless one can choose any one on demand.

Neural Gas

Here a completely different approach was taken. Except of using the distance inside the graph (that doesn't exist at all), a distance in the input space is used to compute the neighborhood function. For that purpose all nodes closer (in the input space) to the sample are counted (see equation 2.18). This result can than be taken as input to the above equations to compute h_{si}.

Growing Neural Gas and Growing Cells

These two methods use a two-step function. The winning node gets the highest, its neighbors a smaller value and all other nodes are not updated at all (see equation 2.25).

Growing Grid

The Growing Grid method uses the city-block metric to compute the distance between the winning and the updating node. Then a Gaussian function (see equation 2.13) is used to compute the final result.

GSOM

The GSOM algorithm is not restricted to one neighborhood function. Any type of relation may be used.

3.2.6 AdaptionNet

The net adaption step is the most divers part of the different methods. This step (that is not part of the original algorithm) is the kernel of all growing approaches (except the Neural Gas). As it modifies the network structure, it also depends on the original structure used.

Neural Gas

As the number of nodes is fixed, no separate growing step is performed. The creation and removal of edges is done in the Adaption step discussed before. This is done because *UpdateNet* is performed every iteration, while the net growing step is only performed after a certain criteria is fulfilled.

Growing Neural Gas

Growing is done by adding a new node at that part of the net, that has the greatest prediction error so far. For that purpose the node with the highest local error measure is taken. The edge to the neighbor node with the highest error value is split and a new neuron is inserted there.

Growing Cells

Here also the node with the highest local error measure is taken. Then the longest edge (measured in the input space) emanating from it is split, retaining the cell structure.

Growing Grid

The growing grid algorithm also uses the node with the highest local error measure to decide where to insert new rows. Then the longest edge emanating form that node is take (as is in the growing cells).

Now this edge (and all parallel ones) is split and a complete row is inserted there.

GSOM

Though rows are inserted too in this algorithm, no other similarities can be found to the previously described methods. Instead of using a local error measure, a complete analysis of the grid is done. Then it is decided whether to insert a new row (and in which dimension) or to add a new dimension (see section 2.6). However, even this very divers model fits into the framework and only the main differences have to be designed new.

3.2.7 Additional steps

Some algorithms require additional steps like removal of cells or a final fine-tuning phase. These steps can be built in very easy, be expanding the above classes.

Growing Cells

As removal of cells is not useful at every iteration, it is done after a net adaption step. This allows to keep the other parts unaffected.

Growing Grid and GSOM

The fine-tuning phase is an application of Kohonens algorithm to the fully grown net. This is done, by applying this algorithm in a second run, with different parameters and without initialization, to the result of the growing process.

Chapter 4

Implementation

In this chapter the implementation of Kohonens algorithm will be discussed. The chapter is split in two part: First the class structure will be explained, to show, what objects are created and how they interact. First the general class structure is presented. Then the data structures are explained, followed by the map operators and some utilities. In the second part of this chapter, a description of the algorithm and their usage is given. Extensions to the basic algorithm discussed in this chapter are described in chapter 5. All classes were implemented in C++ as part of the Analysis++ project.

4.1 Generalized Class structure

First the data model used in the implementation is developed and explained. This model is the kernel for all other SOM algorithms. Therefore it was very important to design the classes with respect to future extensions and modifications.

The classes are presented in three groups. Each group of objects will be explained using UML diagrams to illustrate the dependencies.

SOMMap Data The map is implemented using a grid as underlying structure. To each node, a weight vector is associated. Only the basic operations, like adding and removing a node are available.

SOMMap operators Building upon the map classes, the basic operations required for computing a SOM are defined. In most cases, Kohonens algorithm was implemented as the default. Some of the classes are defined as virtual to gain maximum flexibility.

Basic utilities Some often used classes are presented. They enable the user to set up a complete algorithm and to run simple analyzes.

4.1.1 SOMMap Data

As the map structure may differ from case to case, a representation with maximum flexibility was needed. This goal was achieved by deriving the map from a general undirected graph. The graph allows to add and remove nodes and edges, so that the map-methods can create any structure on demand. The graph also allows easy input/output of the map structure. Additionally this representation allows great flexibility, because any map topology can be represented in such a way.

It was very important, to allow easy exchange of the weight type and the operators used for initialization and computation. This is done by implementing each data-type and operator in a separate class, that can by controlled via an input file.

Overview

The graph is implemented in the `Graph` class. It is associated to a free number of nodes (`Node`) and edges (`Edge`) (see figure 4.1).

In the map class `SOMMap` the basic functionality is implemented by modifiable methods. Using not hard coded routines but associated operator classes, allows easy changes of the maps behavior. As the maps node need additional information, the class `SOMMapNode` is derived from `Node`. It allows additional storage of the weight data. The weight data is derived from the virtual class `SOMWeight` and allows basic computations. It is possible to use any vector space compatible structure as weight. For standard problems, a numeric vector is supported (see section 4.1.3).

Some algorithms need to store additional information for each node or edge. Therefore `Node` and `Edge` are derived from `ListOfObject`. This is a class of Objects, that can hold any class derived from `Object`.

Special purpose maps may be derived from the basis class, as for example the `SOMGrid` class, that additionally allows to define the size of the grid and offers easier handling for computation.

Classes

class Graph General graph that allows adding and removing of nodes and edges. A distance function defining a metric over the graph is available. See section 4.2.1 for a detailed description.

class Node General node class that can store any kind of additional data.

class Edge General edge class that can store any kind of additional data.

class SOMMap The map, containing the data (from `Graph`) and the methods to train the net. See section 4.2.2 for a detailed description.

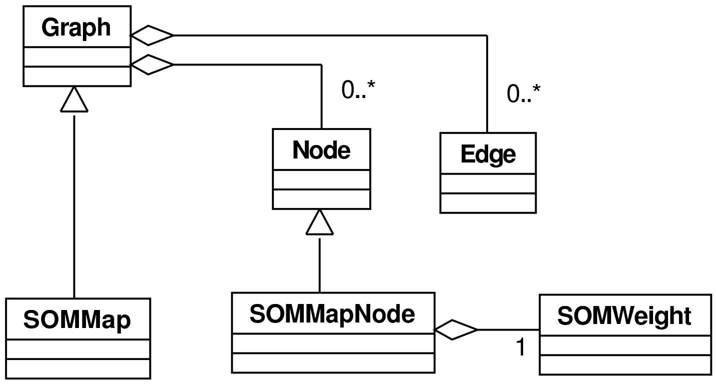

Figure 4.1: Class structure, map data

class SOMMapNode Derived from `Node` by adding a weight.

class SOMWeight Virtual class used for any kind of weight vector. Basis computations like addition, subtraction and multiplication with a factor are supported. Also a norm and a distance are available. See section 4.2.3 for a detailed description.

4.1.2 SOMMap operators

One goal of the design process was to isolate the steps of computation to allow easy exchange and reuse of the methods. To achieve this goal, each operation is done in a own class, that has references to the used sub operators.

The set of classes described here give a basic set up, for all further *Neural Net* algorithms. Mostly Kohonens methods were implemented by default, so that they may be used by derived classes. In some cases, only a virtual class was designed as an outline, because some equivalent variants exist or Kohonens algorithm was to specific.

Overview

Using the same basic steps as described in the algorithms in section 2.1.2, the general algorithm within the framework can be described as follows:

```
Initialization:
// initialize map
   map->Initialize()
      // generate map nodes & edges
      SOMGenerate()
      // fill weights with initial values
```

```
      SOMInitialize()
   end

// train the net
SOMTrainNet()
   // repeat learning until BreakCondition
   while (NOT SOMBreakCondition())

      Selection:
      // get a new sample
      sample = SOMGetNewSample()

      // make a learning step
      SOMAdaption()

         Response:
         // compute best matching node
         bestMatchingNode = SOMBestMatching()

         Adaption:
         // update nodes
         for all nodes
            SOMUpdateNode()
         end
      end
   end
end
```

General Algorithm for SelfOrganizingMaps

Initialization is done only once at the beginning, while Selection, Response and Adaption are repeated for each new input state.

Initialization covers the `SOMGenerate` and the `SOMInitialize` method. `SOMTrainNet` calls the `SOMGetNewSample` to get a new input sample, computes the response using `SOMBestMatching` and finally adapts the weights of the map, using `SOMAdaption` until `SOMBreakCondition` is fulfilled.

The `SOMAdaption` step uses a general `UpdateNode` function, that uses the `SOMNeighborhoodFunction` h_{si} and the `SOMLearningRate` $\alpha(t)$ (see equation 2.11 to 2.14). The growing part is left out at this point, because it doesn't appear in Kohonens model. It is introduced later, when growing methods are discussed.

Classes

The basic classes cover all methods needed to implement the kernel of Kohonens model. The `SOMMap` class introduced before is now extended with the operator classes. See figure 4.2 for a description of the dependencies.

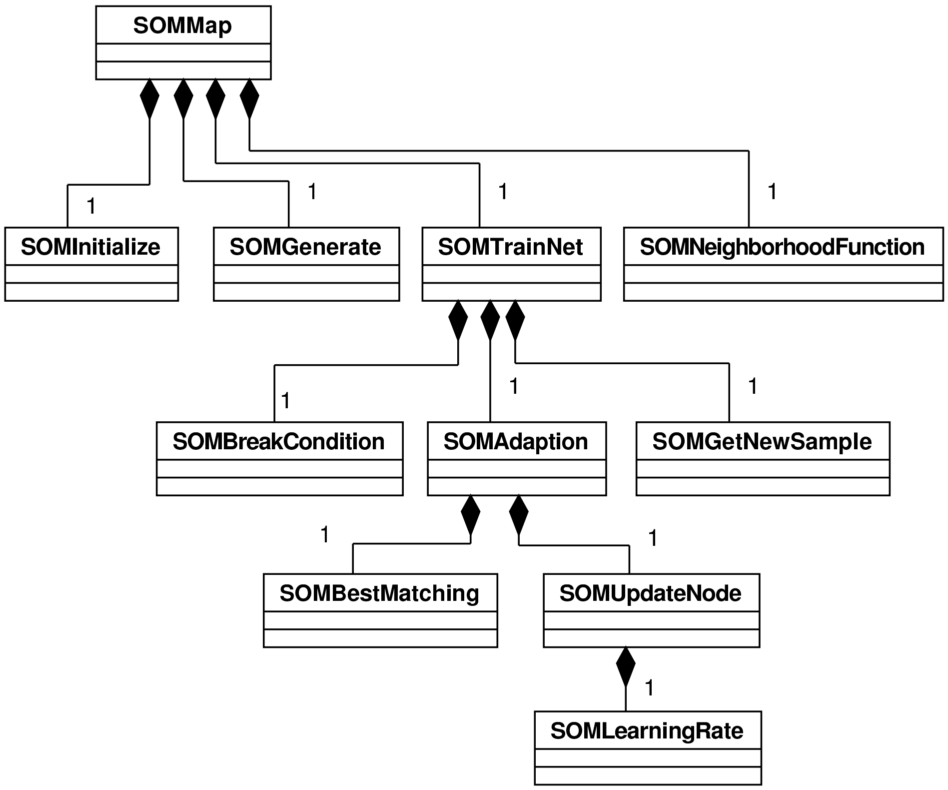

Figure 4.2: Class structure, map operators

virtual class SOMGenerate Virtual Class to generate the map structure (nodes and edges).

virtual class SOMInitialize Virtual class to initializes the nodes of the map with weight data.

class SOMTrainNet This method is used, to run a learning process. In each learning step, a new sample is taken from `SOMGetNewSample` and given to `SOMAdaption` until the `SOMBreakCondition` is fulfilled. See section 4.2.4 for a detailed description.

class SOMBreakCondition Is a test, whether the learning process should be stopped or continue. See section 4.2.5 for a detailed description.

class SOMGetNewSample Returns a new random sample for the learning process.

class SOMAdaption This method is the kernel of the learning process. Given a sample, the *BestMatchingNode* is computed using the `SOMBestMatching` method. Then all nodes are modified, using the `SOMUpdateNode` method. See section 4.2.6 for a detailed description.

class SOMBestMatching Given a sample weight, this method computes the node of the map, with minimal `SOMWeight::Distance`. See section 4.2.7 for a detailed description.

class SOMUpdateNode Modifies a given node, using the `SOMNeighborhoodFunction` and the `SOMLearningRate` to shift it towards the *BestMatchingNode*. See section 4.2.8 for a detailed description.

class SOMLearningRate This operator defines the strength of the learning process. See section 4.2.9 for a detailed description.

class SOMNeighborhoodFunction Computes a real valued weight that describes the influence of the best matching node to a given node. For that purpose usually the `Graph::GraphDistance` is used. See section 4.2.10 for a detailed description.

4.1.3 Basic utilities

This package adds some standard methods to the kernel package, that may be useful, but are not necessary for all derived classes. Most virtual classes defined previously are implemented here.

class SOMNumericVectorWeight A numeric vector class, that is an interface between the numeric data types of the Analysis package and the `SOMWeight` class.

class SOMGenerateGrid Generates a $n \times m \times \ldots$ grid, according to the given size.

class SOMGenerateNodes Generates a fix number of nodes without edges.

class SOMInitializeRandom Initializes the weights of the map-nodes with random values in the range of the input data.

class SOMGetNewSampleInstance Creates a new sample for the learning process from the given instances.

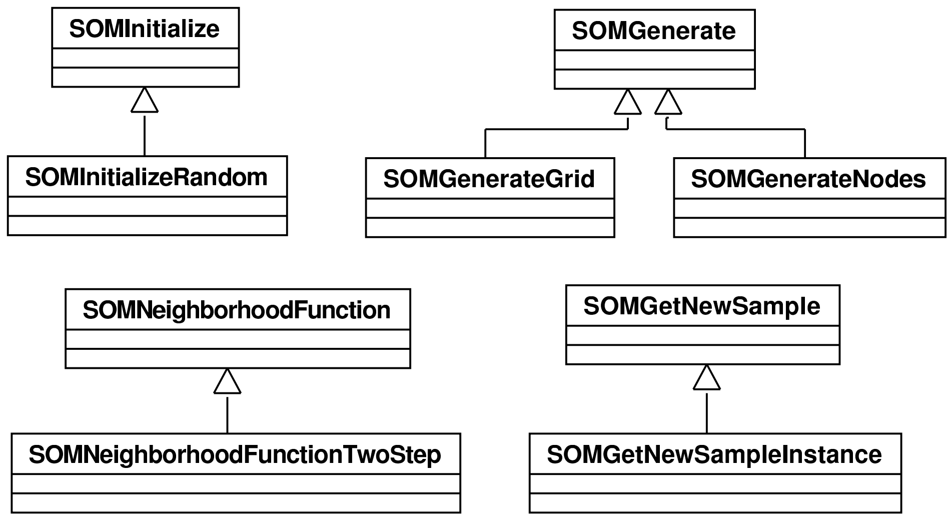

Figure 4.3: Class structure, map utilities

class SOMNeighborhoodFunctionTwoStep Returns α_b for the best matching node, and α_n for its neighbors. Otherwise zero is returned. With this method it is possible to built a simple mexican-hat function if α_n is set to a negative value.

4.2 Component Classes

In this section the most important classes will be discussed in detail. For the operator classes a rough pseudo-code algorithm will be given.

Though the pseudo-code should be self-explanatory in most parts, some preliminary definitions: If a new function is defined, its name is followed by braces and a colon (e.g. `test():`). If a method or function is called, its name is followed by braces (e.g. `test()`). Blocks belonging together are indented.

4.2.1 class Graph

This general graph class is used as a flexible basis for the map. As (undirected) edges can be created between any nodes without restrictions, any kind of structure is possible.

By default the `GraphDistance` method computes the length of the minimal path between two nodes. This distance measure is chosen, because this distance measure is independent form the underlying structure.

From the graphs point of view, this distance measure is defined for the output space and defines the topology of the graph. Other possible choices are, e.g. for a grid, the city block metric.

4.2.2 class Map

The map, containing the data and the methods to train the net. The additional features are a generate operation to set up the net structure, an operation to initialize the node weights, a general neighborhood function and an operation to train the net.
See figures A.1 and A.2 for a description of the input files.

4.2.3 class SOMWeight

Virtual Class that allows basic computations, like addition, subtraction, scalar multiplication and computation of a norm. The norm is used to compute a distance measure.

Seen from the maps view, the `SOMWeight::Distance` method is defined for the input space and compares two objects. In chapter 2.1 the Euclidean distance $\| \bullet \|$ was used.

As any kind of object may be used as an input for the map, the distance method is an important part of the interface between these objects and the map. It defines the topology of the input space that the map should learn. It should be mentioned, that from a mathematical point of view, the choice of the norm method is not completely free, as it has to be a norm in mathematical sense to induce a consistent topology. Furthermore, the distance has to be a vector space. This is necessary to allow computation with its elements.
See figures A.5 and A.6 for a description of the input files.

4.2.4 class SOMTrainNet

This method is used, to run a learning process. In each learning step, a new sample is taken from `SOMGetNewSample` and given to `SOMAdaption` until the `SOMBreakCondition` is fulfilled.

In the preprocessing step, some initial calculations may be done, like a computation of the `GraphDistance` between all nodes to speed up later computation.

```
if (DoPreProcess)
   PreProcess
while (NOT SOMBreakCondition())
   sample = SOMGetNewSample()
   SOMAdaption()
end
```

Algorithm SOMTrainNet

See figures A.7 and A.8 for a description of the input files.

4.2.5 class SOMBreakCondition

Returns 1 if the *BreakCondition* is not fulfilled, 0 otherwise.

The basic implementation is a test, whether $t > t_{max}$. Other possible test would be if a number of nodes has been generated already or if a global error measure is minimized.

```
if (t < t_max)
   result = true
else
   result = false
```

Algorithm SOMBreakCondition

See figures A.9 and A.10 for a description of the input files.

4.2.6 class SOMAdaption

This method is the kernel of the learning process. Given a sample, the *BestMatchingNode* is computed using the `SOMBestMatching` method. Then all nodes are modified, using the `SOMUpdateNode` method.

```
if (DoPreProcess)
   UpdateNode->PreProcess
bmNode = SOMBestMatching()
for all nodes
   SOMUpdateNode(curNode)
end
```

Algorithm SOMAdaption

See figures A.11 and A.12 for a description of the input files.

4.2.7 class SOMBestMatching

Given a sample weight, this method computes the in some way "best matching" node of the map. Usually the weights `Distance()` method is used to determine the minimum distance between the given sample and the weights of the map-nodes.

```
for all nodes
   curDist = node.Weight->Distance(sample.Weight)
   if (curDist < bestDist)
      bestDist = curDist
end
```

Algorithm SOMBestMatching

4.2.8 class SOMUpdateNode

The given node is shifted towards the sample, according to the `SOMNeighborhoodFunction` and the `SOMLearningRate`.

```
hsi = SOMNeighborhoodFunction()
alpha = SOMLearningRate()
factor = alpha * hsi;
if (factor != 0)
   node.Weight = node.Weight +
                 factor * (sample - node.Weight)
end
```

Algorithm SOMUpdateNode

4.2.9 class SOMLearningRate

This operator returns a real value α between 0 and 1, that is used to define the strength of the learning process. If α is 1, the update is done with maximum strength, if it is 0, no update is done at all.

The `SOMLearningRate` $\alpha(t)$ is a (usually) monotonically decreasing function in time. See equation 2.14 for a simple example.

```
result = alpha_start * pow ((alpha_end / alpha_start), t/t_max)
```

Algorithm SOMLearningRate

See figures A.15 and A.16 for a description of the input files.

4.2.10 class SOMNeighborhoodFunction

Computes a real valued weight that describes the influence of the best matching node to a given node. For that purpose, the `Graph::GraphDistance` method is used to determine the distance (in the graph) between the two nodes.

The `Graph::GraphDistance` method defines (together with the structure of the map) the topology of the net. It is a distance function defined for the output space (the map).

A lot of different choices for the neighborhood exist. With this framework it is extremely easy to change the neighborhood function and compare the results. In equation 2.12 and 2.13 two sample definitions are given.

The radius of influence should be time decreasing, to ensure convergence.

The basic set up is a linear decreasing kernel combined with a $[1, 0]$ step function, starting at $t = 0$ width radius `kwidth` and ending with 0 at $t = t_{max}$.

```
dist = map->graphDistance(currentNode, bestMatchingNode)
radius = (1.1*t_max-t)*kwidth / t_max
if (dist < radius)
   result = 1
else
   result = 0
```

Algorithm SOMNeighborhoodFunction

See figures A.17 and A.18 for a description of the input files.

Chapter 5

Extensions

Based upon the implementation of Kohonens algorithm in chapter 4 some extensions to his model are discussed. They are set into relation and use the advantages of the object-oriented design to minimize and ease later modifications. In opposite to the previously described methods, all methods presented in this chapter use a flexible net topology. This is done, by adding nodes and edges on demand. Though, the methods differ a lot in how this is done, they were implemented inside the framework with only a few modifications. Especially a new growing-step had to be inserted, that can be modified on demand. As an criteria where to insert new nodes or edges, usually a local error measure is used. This error measure is stored locally inside the nodes and removed afterwards. As of the great flexibility of the framework and great brought of different methods, these extensions can only act as samples for further development.

5.1 Class structure

Extending the basic class structure, each algorithm is described in his modifications and extension. UML diagrams will show, how the object-oriented design minimizes the need for additional and modified classes.

5.1.1 Neural Gas

This set of methods allow an implementation of the *Neural Gas* algorithm. As most of the original algorithm was reused, only three operators had to be modified. They were derived from the original classes and reuse parts of them, so that only a few differences had to be implemented. These methods are also the basis for the *Growing Neural Gas* algorithm.

Overview

The neural gas algorithm has no pre-defined map structure, but a fix number of nodes. Edges are added and removed using the methods available from the `Graph` class, the initial set up is done using the `SOMGeneratenodes` and `SOMInitializeRandom` classes from the utility package.

To compute the *Learning Rate*, the number of nodes closer to the input sample has to be computed. This is done by preprocessing the nodes and storing this temporal information in the nodes `SetOfObjects`. This operation is independent from the original one, and does not use any of its components.

As adaption requires not only the best, but also the second best matching node, a modified version of `SOMBestMatching` is used. It is only an extension to the original operation in that sense, as it uses the same algorithm, but also computes the second best matching node, needed for further computations.

The adaption class itself reuses the original adaption operator and adds some new adaption steps, necessary for creating new edges and removing old ones.

Classes

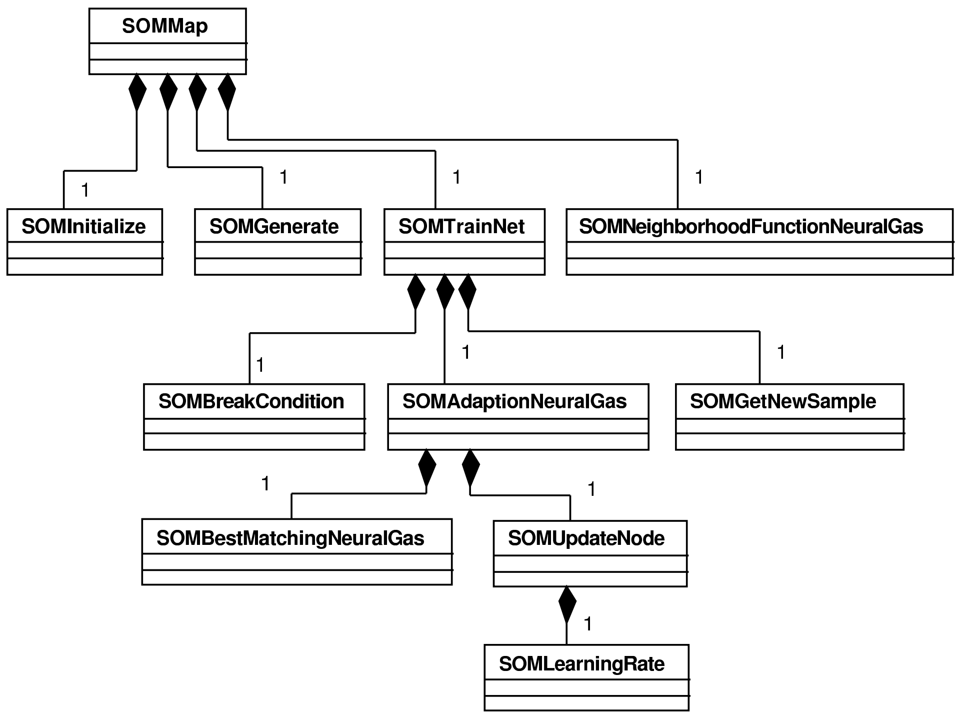

Figure 5.1: Class structure, Neural Gas

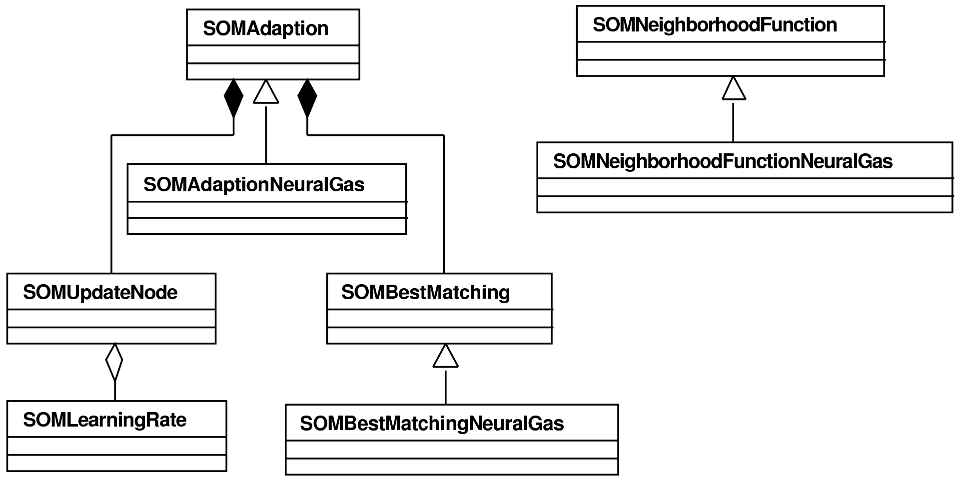

Figure 5.2: Class structure detailed, Neural Gas

Figure 5.1 shows the standard relations for the Neural Gas implementation. In figure 5.2 the new relations are shown in detail.

class SOMAdaptionNeuralGas Updates the nodes using the `SOMUpdateNode` method. Additionally an edge between the best and the second best matching node is creates. Old edges are removed. See section 5.2.1 for a detailed description.

class SOMBestMatchingNeuralGas Computes the best and the second best matching node. See section 5.2.2 for a detailed description.

class SOMNeighborhoodFunctionNeuralGas Computes the *NeighborhoodFunction* according to equation 2.18. See section 5.2.3 for a detailed description.

5.1.2 Growing Neural Gas

This is the first complete growing algorithm implemented in the framework. It is used to show the benefits of the object-oriented approach. The kernel, referring to Kohonens algorithm has been reused, while the new growing steps were added seamless into the system.

Overview

The *Growing Neural Gas algorithm* is a classical growing algorithm in that sense, as it splits the learning process in two parts: A weight adaption step, as known from Kohonen, and a net adaption step, that is performed frequently to update the net structure. While the *Neural Gas* algorithm only added new edges, this one is also able to create new nodes.

The new learning step is covered by the new `SOMTrainNetGrowing` operator, that extends the standard operator. The learning step itself is implemented in the `SOMAdaptionNetGNG` class, which is derived from the virtual `SOMAdaptionNet` class. This allows easy exchange of this step, by simply creating a new derived class. For example, the criteria for adding a new node may be changed, so that an external error measure can be considered.

The adaption step for the weights extends the `SOMAdaptionNeuralGas` operation, but reuses most of it. Only a new temporal error measure is computed, that is later used to decide where to add new neurons.

The *break condition* was also modified, to stop the growing process, if a predefined number of nodes has been created, or a upper bound of iterations was reached.

The `SOMLearningRateTwoStep` operator was used from the utilities package and the `SOMBestMatchingNeuralGas` operator was reused from the NeuralGas algorithm.

Classes

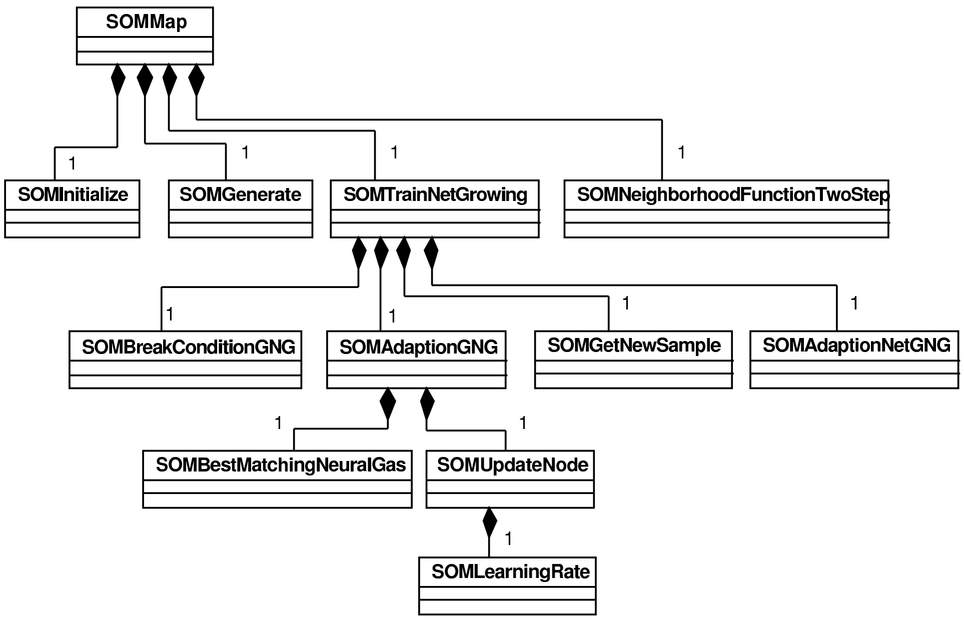

Figure 5.3: Class structure, Growing Neural Gas

Figure 5.3 shows the standard relations for the Growing Neural Gas implementation. In figure 5.4 the new relations are shown in detail.

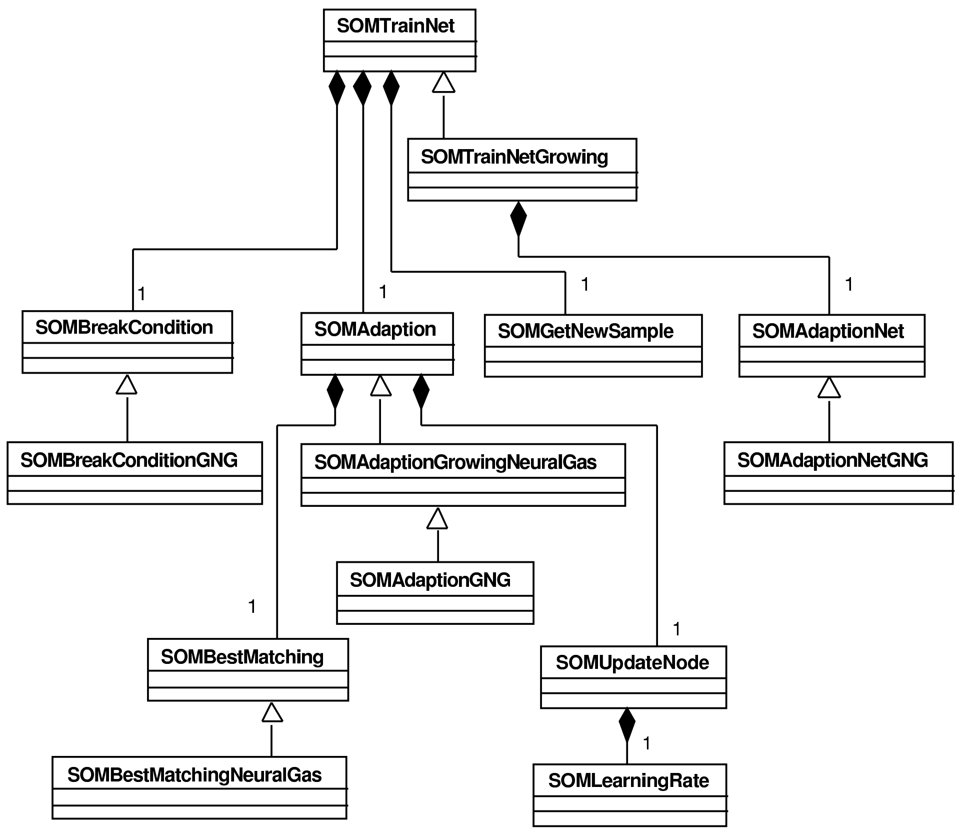

Figure 5.4: Class structure detailed, Growing Neural Gas

class SOMTrainNetGrowing Performs the basic loop needed to train the net. For each sample, the weights of the nodes, and after a fix amount of iterations, the net structure is updated. See section 5.2.4 for a detailed description.

class SOMBreakConditionGNG Returns true, if a predefined number of nodes is created so far, or enough learning steps are done.

class SOMAdaptionGNG Updates the nodes weights and creates new edges between the two best matching nodes. A local error measure is computed. See section 5.2.5 for a detailed description.

class SOMAdaptionNet Virtual Class that allows to perform a net growing step.

class SOMAdaptionNetGNG Compares the local error measures to decide, where to add a new neuron. See section 5.2.6 for a detailed description.

5.1.3 Growing Cells

As an outlook, now the changes needed to implement the *Growing Cells* algorithm will be discussed. It is shown, that only the real difference to the *Growing Neural Gas* algorithm, the handling of adding a new neuron, has to be considered. The rest of the algorithm uses the previous discussed methods.

Overview

As this algorithm needs a special map structure (k-dimensional simplices), a new class `SOMVertices` was derived from `SOMMap`. Also the initial generation of the map needed to be modified, to create a single vertice at start up.

The map is trained with the previous discussed `SOMTrainNetGrowing` method, by only changing the adaption steps.

The adaption step, performed in this algorithm, is mostly the same as in the Growing Neural Gas, except that no age information is considered, and therefor no old edges are removed. This allows to reuse great parts of the previous adaption methods.

When adding a new neuron in the `SOMAdaptionNetGCells` method, the cell structure (k-dimensional simplices) has to be prevented. This is done within the `SOMVertices` class.

Classes

Figure 5.5 shows the standard relations for the Growing Cells implementation. In figure 5.6 the new relations are shown in detail.

class SOMVertices Derived from `SOMMap` to support a fixed dimensioned network structure of k-dimensional simplices. For that purpose special operations for adding edges and removing nodes or edges are supported.

class SOMAdaptionGCells Updates the nodes and computes a local error measure for the winning node. See section 5.2.7 for a detailed description.

class SOMAdaptionNetGCells Compares the local error measures to decide, where to add a new neuron. This is done preserving the network dimensionality. See section 5.2.8 for a detailed description.

5.2 Component Classes

For an easier understanding, the most important components will be described here in detail. Pseudo-code algorithms are used to explain the methods and show the dependencies inside the code.

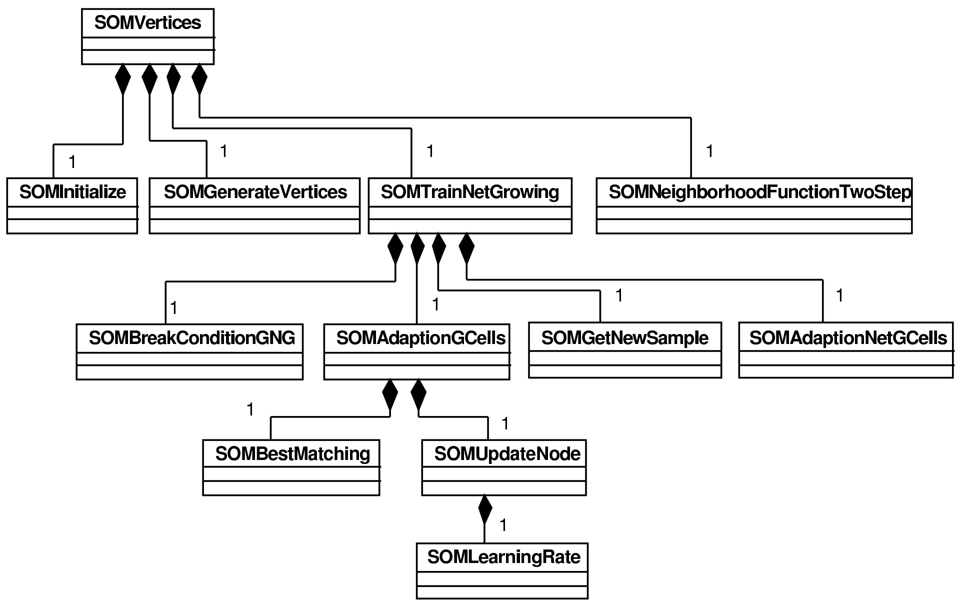

Figure 5.5: Class structure, Growing Neural Cells

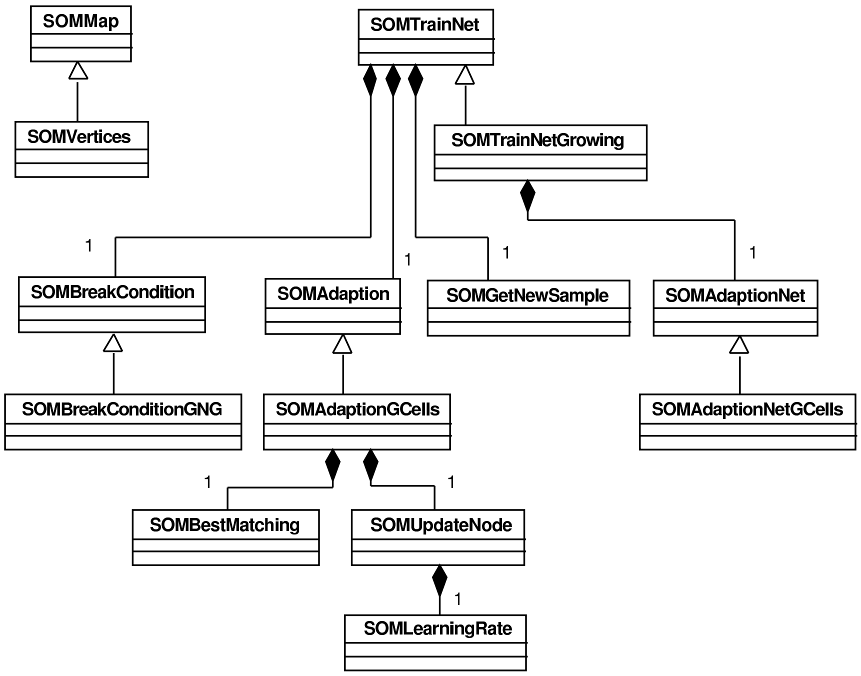

Figure 5.6: Class structure detailed, Growing Neural Cells

5.2.1 class SOMAdaptionNeuralGas

Updates the nodes using the `SOMUpdateNode` method, and additionally creates an edge between the best and the second best matching node. All edges with an age greater then `age_max` are removed. The additional `age` information was stored temporal in the nodes and only used here, so that the objects integrity is kept.

```
// PreProcessing for the NeighborhoodFunction is necessary
DoPreProcess = true
// call the original SOMAdaption operation
SOMAdaption()
// add new edge
AddNewEdge(bestNode, bestNode2)
// remove old edges
UpdateEdges():
   max_age = max_age_start *
             pow((max_age_end / max_age_start), t/t_max)
   for all edges
      curEdge->age++
      if (age > max_age)
         DelEdge(curEdge)
   end
end
```

Algorithm SOMAdaptionNeuralGas

5.2.2 class SOMBestMatchingNeuralGas

Using the same algorithm as in `SOMBestMatching`, the best and the second best matching node are computed.

```
for all nodes
   curDist = Node.Weight->Distance(sampleWeight)
   if (curDist < bestDist)
      bestDist2 = bestDist
      bestDist = curDist
   if (curDist < bestDist2)
      bestDist2 = curDist
end
```

Algorithm SOMBestMatchingNeuralGas

5.2.3 class SOMNeighborhoodFunctionNeuralGas

Computes the *NeighborhoodFunction* according to equation 2.18. k, the number of nodes closer to the input sample is computed for all nodes in a preprocessing step.

In the preprocessing step first for all nodes the distance to the input sample is computed, and then the nodes with a shorter distance are counted. The result is stored as a temporal information in each node. This is possible, through the implementation of the nodes, as `SetOfObjects`, a kind of container class, that can hold any kind of objects.

```
// k is the number of nodes closer to the
// sample and is computet in a preprocessing step
k = Node->k

lambda = lambda_start *
         pow((lambda_end / lambda_start), t/t_max)
result = exp(-(k / lambda))
```

Algorithm SOMNeighborhoodFunctionNeuralGas

5.2.4 class SOMTrainNetGrowing

This method is used, to run a learning process. In each learning step, a new sample is taken from `SOMGetNewSample` and given to `SOMAdaption` until the `SOMBreakCondition` is fulfilled. If the number of learning steps done so far is a multiplier of a given number, `SOMAdaptionNet` is called to modify the network structure.

```
if (DoPreProcess)
   PreProcess
while (NOT SOMBreakCondition())
   sample = SOMGetNewSample()
   SOMAdaption()

   if (NumberOfLearningSteps modulo t_AdaptionNet = 0)
      SOMAdaptionNet()
end
```

Algorithm SOMTrainNetGrowing

5.2.5 class SOMAdaptionGNG

Reuses the `SOMAdaptionNeuralGas` class and adds a local error measure to the winning node. This error measure describes the prediction error of the best matching node, to the sample given. Summing up his errors, an average prediction error is computed, that allows to find regions with too few nodes. This error measure may be changed, to find regions, satisfying other goals.

```
// call the original SOMAdaption operation
SOMAdaption
// add new edge
AddNewEdge(bestNode, bestNode2)
// remove old edges
UpdateEdges():
   SOMAdaptionNeuralGas::UpdateEdges()
   // remove nodes with no edges left
   if (|GetNeighboredEdges(curNode)| = 0)
      DelNode(curNode)
// update weight of winning node
UpdateWinning():
   bestNode.Error = bestNode.Error +
                    Distance(bestNode.Weight, sample)
end
// call UpdateEdges from super class
SOMAdaptionNeuralGas::UpdateEdges()
UpdateWeights():
   for all nodes
      // update error
      curNode.Error = (1-beta) * curNode.Error
   end
end
```

Algorithm SOMAdaptionGNG

5.2.6 class SOMAdaptionNetGNG

Compares the local error measures to find a node with the greatest prediction error. A new neuron is added between this node and its neighbor with the highest error. The connections are adapted accordingly.

```
q = node with maximum error value of all nodes
f = node with maximum error value of all neighbors of q
AddNode(r)
```

```
r.Weight = (q.Weight + f.Weight) / 2
DelEdge(q,f)
AddEdge(q,r)
AddEdge(r,f)
q.Error = (1-alpha) * q.Error
f.Error = (1-alpha) * f.Error
```

Algorithm SOMAdaptionNetGNG

5.2.7 class SOMAdaptionGCells

Derived from `SOMAdaptionGNG` most parts were reused. Only the part, adding and removing edges was left out. The computation of the error measure was kept.

```
// call the original SOMAdaption operation
SOMAdaption
// compute and update the weights
SOMAdaptionGNG::UpdateWinning()
SOMAdaptionGNG::UpdateWeights()
```

Algorithm SOMAdaptionGCells

5.2.8 class SOMAdaptionNetGCells

Compares the local error measures to decide, where to add a new neuron. This is done preserving the network dimensionality.

```
q = node with maximum error value of all nodes
f = node with longest edge emanating from q
r = SplitEdge(q,f):
   r.Weight = (q.Weight + f.Weight) / 2
   // create new connections to preserve the network structure
   neighbors = (nodes having a common edge with q and f)
   for all nodes in neighbors
      AddEdge(curNode,r)
   end
end
// update error values
neighbors = GetNeighboredNodes(r)
sumError = 0
```

```
for all nodes in neighbors
   curNode.Error = (1-alpha / |neighbors|) * curNode.Error
   sumError = sumError + curnode.Error
end
r.Error = sumError / |neighbors|
```

Algorithm SOMAdaptionNetGCells

Chapter 6

Examples

In this chapter some examples are presented to show the ability of the model to deal with the same problem in different ways, only by simple changes in the input files. Also the power of the algorithm to deal with different goals and input data is illustrated. First, the examples are discussed from a general point of view, followed by a detail explanation of the different solutions. The input files for each example can be found in the Appendix. Three examples are given:

Simple Sample An introductory sample to illustrate the mapping of the net.

Animals 16 animals are described by 13 boolean attributes. The goal is to group similar animals together.

Iris flowers Three species of iris flowers are described by four parameters. The goal is to separate the species and enable the net to classify new flowers correctly.

6.1 A simple sample

This sample is the standard test for equal distributed input data. It uses the read in lower and upper bound of the map, to generate randomly input data from the hypercube, limited by these two points. For easier representation, a 2 dimensional square was used. The result should be an equally distributed two dimensional net (see figure 6.1).

6.1.1 Kohonen Grid

Kohonens algorithm, applied on a 5×5 grid, resulted in a flat grid, with equal distanced nodes. The voronoi region for each node is nearly equal in size, as can be seen in figure 6.2. Here the weight vectors of the grid were projected to the input space.

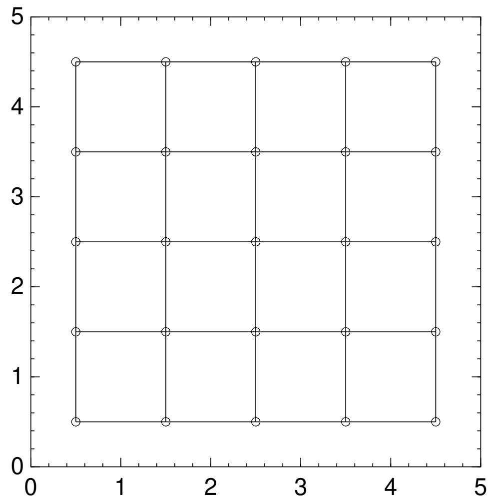

Figure 6.1: Simple sample, Perfect Grid

The parameters for this test were ($LowerBound = [0,0]$, $UpperBound = [5,5]$, $kwidth = 3$, $t_{max} = 2000$, $\alpha_{start} = 0.6$, $\alpha_{end} = 0.05$). See the input files in Appendix B.1.1 for further details.

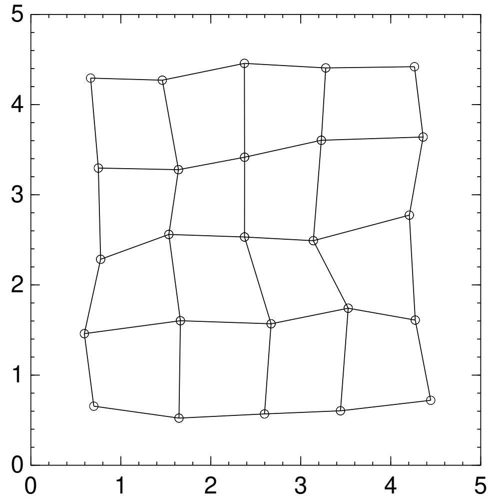

Figure 6.2: Simple sample, Kohonen Grid

6.1.2 Neural Gas

The neural gas algorithm was initialized with 20 randomly placed nodes. The result is a triangulation of the input space. See figure 6.3. The voronoi regions for each node are of equal size as before.

The parameters for this test were ($LowerBound = [0,0]$, $UpperBound = [5,5]$, $NewNodeCount = 20$, $\lambda_{start} = 5.0$, $\lambda_{end} = 0.01$, $t_{max} = 4000$, $age^{max}_{start} = 20$, $age^{max}_{end} = 200$, $\alpha_{start} = 0.5$, $\alpha_{end} = 0.005$). See the input files in Appendix B.1.2 for further details.

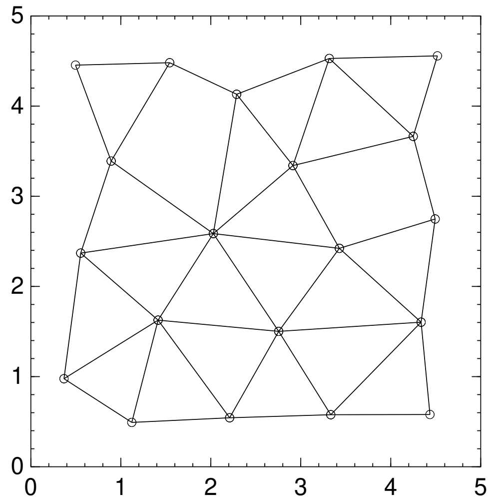

Figure 6.3: Simple sample, Neural Gas

6.1.3 Growing Neural Gas

The growing neural gas was initialized with 2 neurons. Training was repeated until 20 nodes were generated. The result are equal distributed nodes, grouped as squares (see figure 6.4). This shows the ability of the method, to find the correct dimension of the input data and to create the network structure as needed.

The parameters for this test were ($LowerBound = [0,0]$, $UpperBound = [5,5]$, $NewNodeCount = 2$, $\alpha_{winning} = 0.05$, $\alpha_{neighbors} = 0.005$, $t_{max} = 10000$, $t_{net} = 400$, $age^{max}_{start} = 311$, $age^{max}_{end} = 311$, $\alpha_{start} = 1.0$, $\alpha_{end} = 0.5$, $maxNodes = 20$, $\epsilon = 0.5$). See the input files in Appendix B.1.3 for further details.

6.2 Animals

This example was also used by Kohonen [16]. It shows, how a self organizing map can be used, to project high-dimensional binary data, to a two dimensional grid or other topologies.

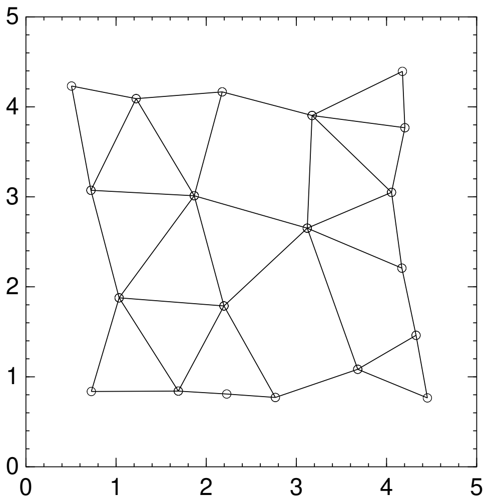

Figure 6.4: Simple sample, Growing Neural Gas

Each animal is described by 13 different binary attributes such as *small* (*yes/no*) or *fly* (*yes/no*). These 13 descriptions were taken as a 13-dimensional binary vector and presented to the net. Kohonen used the dot-product to describe similarity between two animals. In this framework, the distance is computed as the norm of the difference vector. If binary data is used, this is the number of attributes with different values. So without any modifications, the original setup could be used.

The input data has the following format:

```
                         f
                         e
          m          h   a
        s e    2 4   o   t           c
        m d    l l h o m h  h     s  l
        a i b  e e a v a e  u r f w  a
        l u i  g g i e n r  n u l i  s
        l m g  s s r s e s  t n y m  s

hen     1 0 0  1 0 0 0 0 1  0 0 0 0  1
dove    1 0 0  1 0 0 0 0 1  0 0 1 0  1
duck    1 0 0  1 0 0 0 0 1  0 0 1 1  1
goose   1 0 0  1 0 0 0 0 1  0 0 1 1  1
owl     1 0 0  1 0 0 0 0 1  1 0 1 0  1
hawk    1 0 0  1 0 0 0 0 1  1 0 1 0  1
eagle   0 1 0  1 0 0 0 0 1  1 0 0 0  1
```

fox	0 1 0	0 1 1 0 0 0	1 0 0 0	2
dog	0 1 0	0 1 1 0 0 0	0 1 0 0	2
wolf	0 1 0	0 1 1 0 1 0	1 1 0 0	2
cat	1 0 0	0 1 1 0 0 0	1 0 0 0	2
tiger	0 0 1	0 1 1 0 0 0	1 1 0 0	2
lion	0 0 1	0 1 1 0 1 0	1 1 0 0	2
horse	0 0 1	0 1 1 1 1 0	0 1 0 0	3
zebra	0 0 1	0 1 1 1 1 0	0 1 0 0	3
cow	0 0 1	0 1 1 1 0 0	0 0 0 0	3

Animal data set

As the input data is very high dimensional, no direct visual presentation is possible. Instead of using the weight-vectors, the position of the nodes inside the net can be taken to create a visualization. Although Fritzke suggested a method to order the nodes with respect to their relative weight distance, only the graph structure will be shown here.

6.2.1 Kohonen Grid

This is the same approach Kohonen used. A 5×5 grid was used to learn the neighborhood relations in the input data. As there are more nodes than samples, some nodes represent situations, that lie in between the given samples. One can see, that similar animals are grouped together, while more different are more fare away. As the grid is a homogeneous structure, no clusters can be seen ad hoc. Only after taking the distances of the weight vectors into account, some clusters can be found.

As the net is two dimensional, different neighborhood relations can be found. E.g. the eagle is close to tiger and wolf, but also to owl and hawk. This reflects the fact, that a eagle is quite big and hunting, but also a bird. See figure 6.5. Each node of the grid is named with the animals, that are mapped onto it. This allows a reduction of the input space dimension (13) down to two, so that dependencies can be shown very easy.

It is also easy to see, how the three clusters are grouped together. One cluster contains all birds (dove, hen, duck, goose, owl, hawk and eagle). At its lower border, other hunting animals are grouped together (cat, fox, dog, wolf, tiger and lion). The other animals (horse, zebra and cow) are grouped together in the upper right.

The parameters for this test were ($LowerBound = [0, 0, ..]$, $UpperBound = [1, 1, ..]$, $kwidth = 2$, $t_{max} = 1000$, $\alpha_{start} = 0.5$, $\alpha_{end} = 0.005$). See the input files in Appendix B.2.2 for further details.

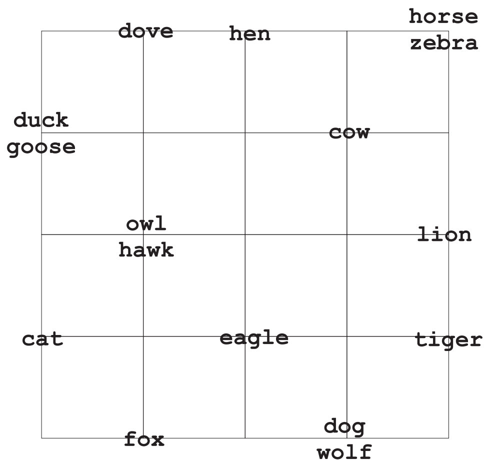

Figure 6.5: Animals sample, Kohonen Grid

6.2.2 Neural Gas

This approach is different form Kohonens, because the Neural Gas tries not to reduce the dimension of the input space to a given one. Now 16 nodes (one for each animal) were used instead of 25 as before. In opposite to the predefined grid, similar nodes are connected, while different ones are separated. This results in a number of clusters, with in some sense similar animals. As mentioned above, one may try to place them on a plane, with respect to their distance in the input space, but for our purpose, a simple illustration is sufficient (see figure 6.6). In this sample, four clusters have been formed, that are similar to Kohonens results. But in opposite to his results, the clusters are found from the algorithm without later interpretation.

The parameters for this test were ($LowerBound = [0, 0, ..]$, $UpperBound = [1, 1, ..]$, $NewNodeCount = 16$, $\lambda_{start} = 5.0$, $\lambda_{end} = 0.01$, $t_{max} = 1000$, $age^{max}_{start} = 10$, $age^{max}_{end} = 100$, $\alpha_{start} = 0.5$, $\alpha_{end} = 0.005$). See the input files in Appendix B.2.3 for further details.

The number of connections (and in consequence the number of clusters) is influenced by age^{max}. If its value is low, fewer connections are created. In the above sample, the animals are grouped with respect to their similarity. E.g. eagle and hen differ only in three criterias, while eagle and dove differ in four attributes. (This may seem a bit strange, but Kohonens input data described the eagle as not flying). On the other hand, the hen and the dove which only differ in one argument are separated. This result may change in an other learning process and is influenced by the initial setup. As nodes are positioned at the beginning of the learning process and edges are only created between the best and the second best matching node, such results can occur. To avoid such in some kind random results, more passes need to

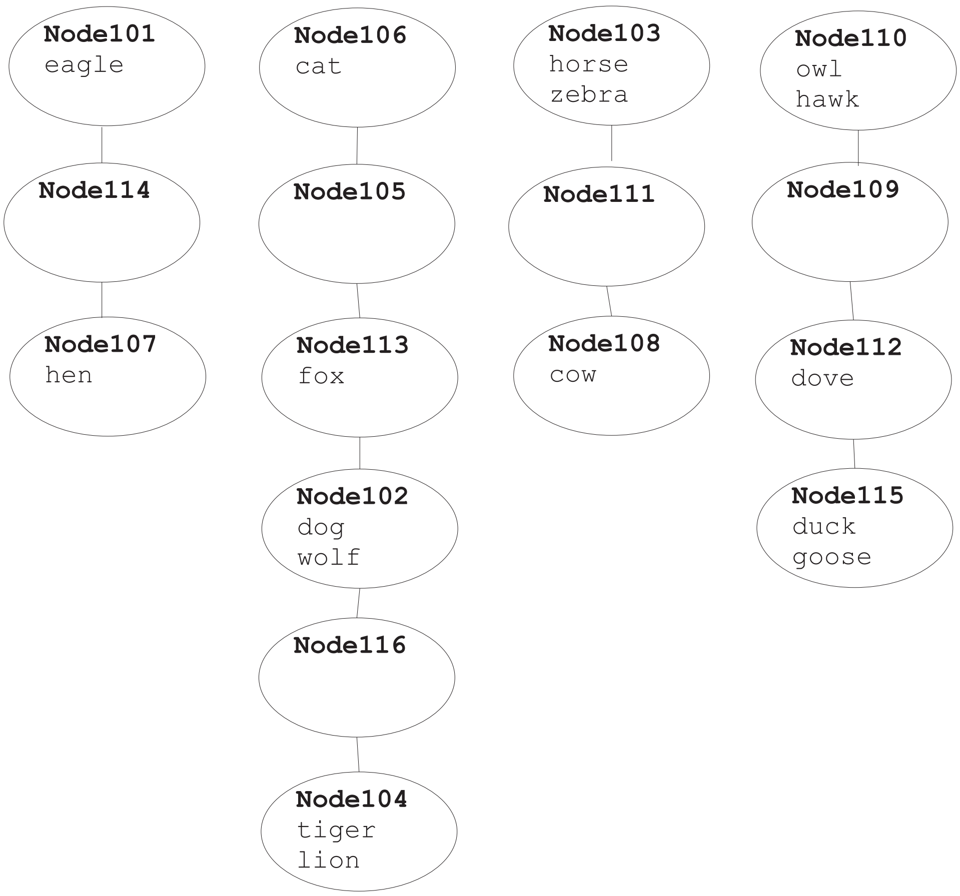

Figure 6.6: Animals sample, Neural Gas

be compared. The other animals are grouped accordingly.

6.3 Iris flowers

This data set was first used by fisher in 1936 as an example for discriminant analysis. The data report four characteristics *(sepal width, sepal length, pedal width and pedal length)* of three species of Iris flowers. 100 samples were given.

The method used by fisher, a discriminant analysis, resulted in between 1 and 3 misclassified observations. The goal of this approach was to find a segmentation of the input space, so that each segment describes only one species of flowers. Now the Neural Gas and a Growing Neural Gas algorithms were used. Both methods were trained without the class information. Nevertheless both produced a good fitting net, that was able to approximate the classes accurate.

Figure 6.7 shows the distribution of the input data set. It is easy to see, that one class is isolated, but the other two are close to each other. The fourth parameter was omitted in this plot, because it has only a small influence for the classification.

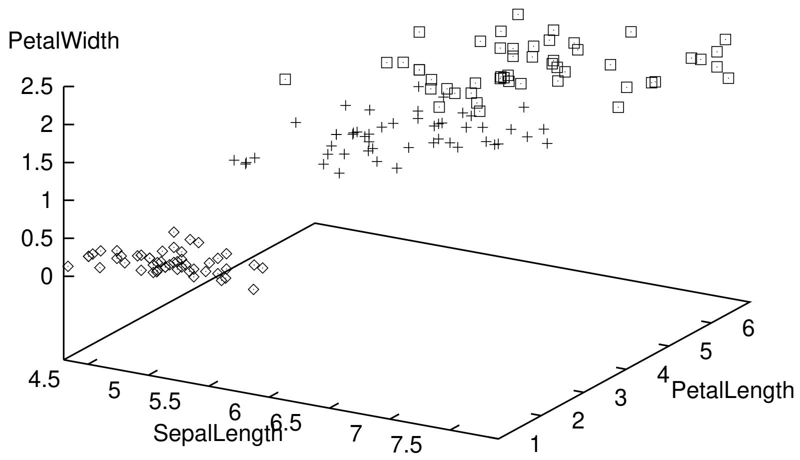

Figure 6.7: Iris input distribution

The different methods used, resulted in between 2 to 3 misclassified samples as fishers original analysis. Classification is done by assigning each node a class, which is the one, most samples mapped onto it are from. It is a strength of the methods, that unsupervised learning resulted in a very good prediction. This shows, that the distinction between the classes is not arbitrary at all and the net is able to find even small differences in the data by itself.

Nevertheless supervised learning would bring better results. New neurons could be inserted in regions where a lot of wrong classifications occur. This would lead to a small number of nodes in the center of each class, but a dense net at the borders. These modifications are very easy to make inside the framework, as only the computation of the local error variable has to be changed.

6.3.1 Neural Gas

The net was initialized with 10 randomly placed nodes. Then the original Neural Gas algorithm was used to train the net. As no special modifications were necessary, the tests were very fast implemented. Only the input data had to be set up accordingly. It was also very easy to run several test, with different parameters, by only changing the input files. This showed, that 10 nodes are required as a minimum, and that more nodes result only in a slightly better result. For the final result with 10 nodes see figure 6.8.

One class (0.5) is separated from the others, while the other two classes are connected. Nevertheless each nodes belongs to one class. Only two nodes contain samples from more than one class. If one takes into account that learning was done without the class information, this result is rather good.

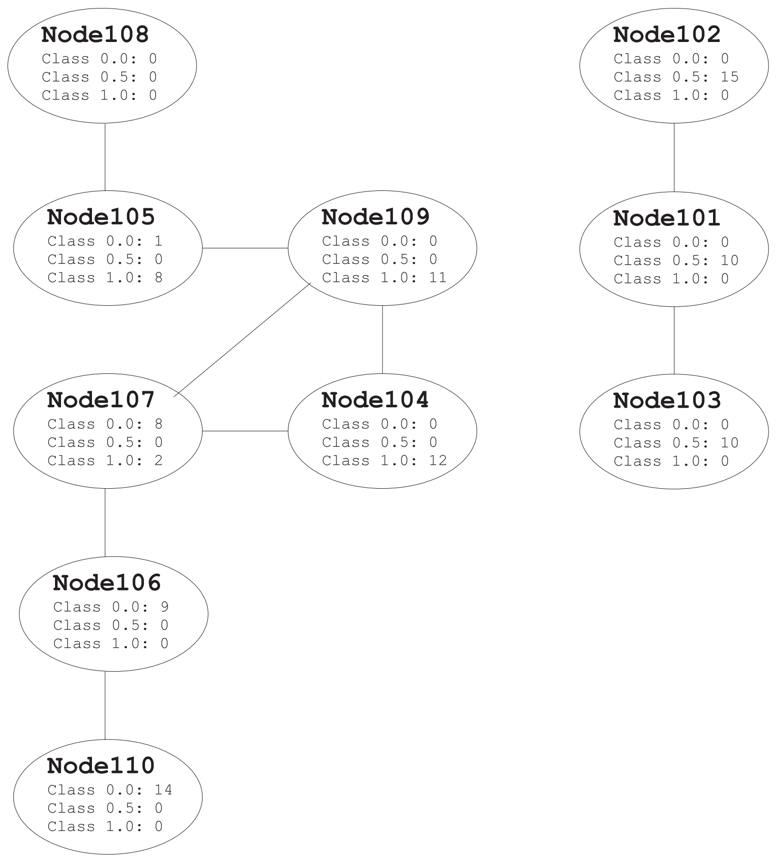

Figure 6.8: Iris sample, Neural Gas

The parameters for this test were ($LowerBound = [5, 2, 0, 0]$, $UpperBound = [8, 4, 7, 3]$, $NewNodeCount = 10$, $\lambda_{start} = 5.0$, $\lambda_{end} = 0.01$, $t_{max} = 2000$, $age_{start}^{max} =$

10, $age_{end}^{max} = 100$, $\alpha_{start} = 0.5$, $\alpha_{end} = 0.005$). See the input files in Appendix B.3.2 for further details.

6.3.2 Growing Neural Gas

Starting with 2 nodes, the net grew until 20 nodes were generated. This allowed a sufficient resolution of the input space, especially at the borders between the classes. The topography of the final net is similar to the one created from the Neural Gas. The center where a lot of input data sites, was mapped onto a two dimensional structure, while the outer parts are one dimensional. See figure 6.9.

The class "0.5" is completely separated. The two classes 0 and 1 are connected, but rather good described by different nodes. Only two nodes do one misclassification each.

The parameters for this test were ($LowerBound = [5, 2, 0, 0]$, $UpperBound = [8, 4, 7, 3]$, $NewNodeCount = 2$, $\alpha_{winning} = 0.05$, $\alpha_{neighbors} = 0.005$, $t_{max} = 10000$, $t_{net} = 200$, $age_{start}^{max} = 111$, $age_{end}^{max} = 111$, $\beta = 0.005$, $\alpha_{start} = 1.0$, $\alpha_{end} = 0.5$, $maxNodes = 20$, $\epsilon = 0.5$). See the input files in Appendix B.3.3 for further details.

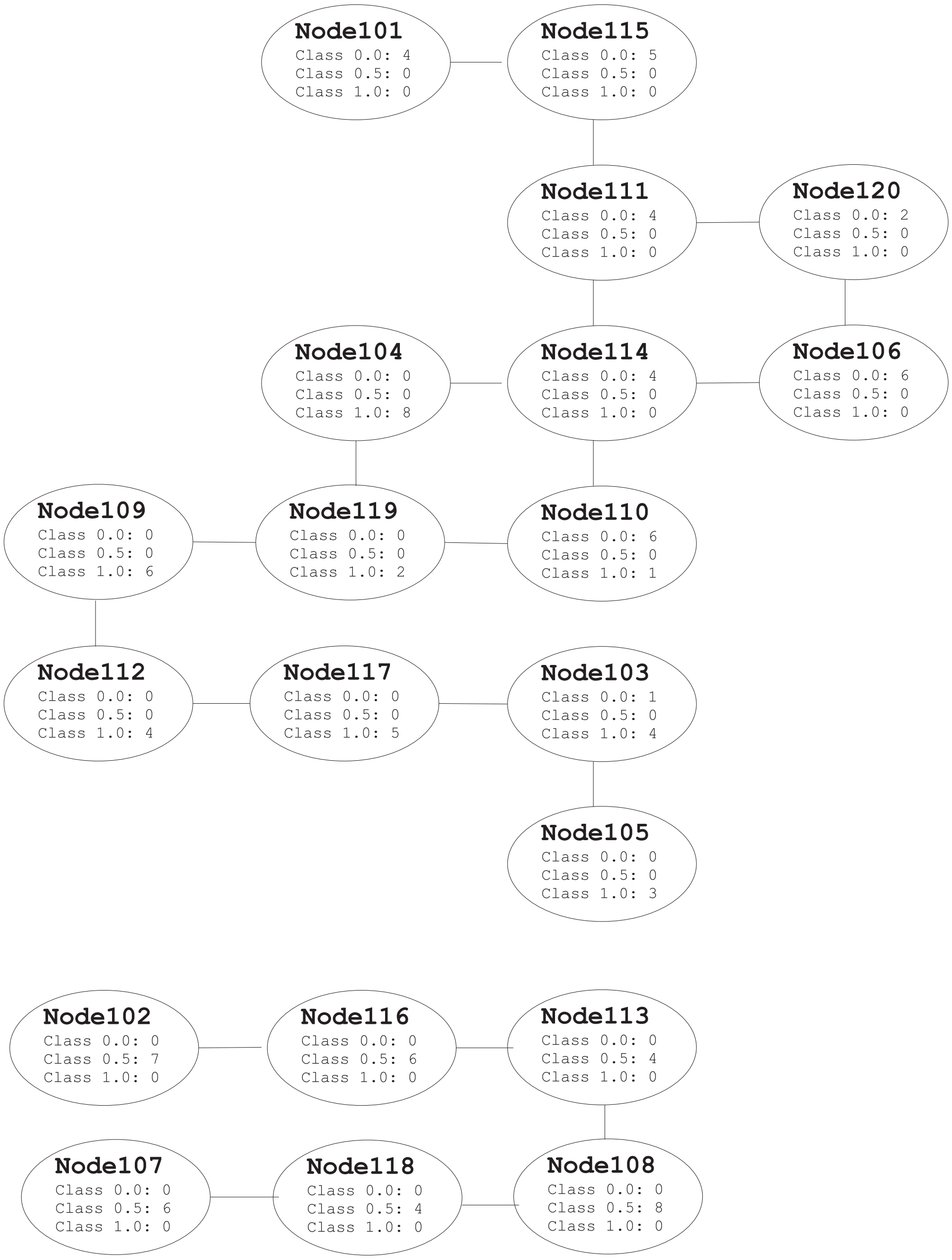

Figure 6.9: Iris sample, Growing Neural Gas

Chapter 7

Outlook

In this work the main emphasis was to develop a general framework for Neural Maps. This was achieved by analyzing different algorithms and creating a common object structure. As a matter of time, only a few algorithms and methods could be implemented. Nevertheless, the framework is flexible enough to allow easy extension. In the following paragraphs I will try to give an outlook on some extensions and variants that may be built using the concepts that have been developed so far.

7.1 Extensions

Two different directions of extending the basic framework are possible. On the one hand, one may implement new algorithms, or combine the available algorithms with other approaches. On the other hand, it could be sufficient, to use new methods (like a new error-function or a different distance measure) to fit the available algorithms to personal needs. Both approaches are supported by the framework through its object oriented design.

In the current implementation, the core classes contain the general graph parts (that build up the map), as well as the basic algorithm structure (Initialize, Generate, TrainNet, ..). All of these classes refer to other classes, to enable easy exchange and extension of each part. E.g. the Growing Neural Gas algorithm introduces a new step, where neurons are added to the net. This is done, by deriving a new class `SOMTrainNetGrowing` from `TrainNet` that performs (in addition to the original steps) an net adaption step performed in the (`SOMAdaptionNet`) class.

7.1.1 New algorithms

As mentioned above, only selected methods are implemented yet. Beside Kohonen's core algorithm, two extensions (Neural Gas and Growing Neural Gas) are available. They reuse the main parts of the original algorithm and extend them with some new modules (see chapter 3).

Extending the framework is a simple task, as a lot of components should be reusable. Especially the components described in chapter 4 that build the kernel of the framework, will occur in most Neural Map implementations. Also the data handling routines, as provided by the *Analysis++* package, will help to setup new algorithms.

Development of new algorithms

In chapter 2 several growing variants of Kohonen's algorithms have been presented. Beside those already implemented within the framework, the *Growing Cells* algorithm, seems very promising. The needed class structure is presented in section 5.1.3. As one can see there, only two new methods (a new adaption step and a new net adaption step) are necessary! Nevertheless, as a new topology for the output-space is used, a new class for the map itself that provides additional operations, is needed (derived from the original `SOMMap` class). But this doesn't concern the algorithm directly.

Another very interesting algorithm is the *Growing Grid*. Here a similar situation as with the Growing Cells occurs: The necessary modification inside the algorithm classes are small, but a new map class is necessary that supports add and remove of rows.

Interaction with other algorithms

Another interesting goal for future development will be the possible interaction of the different methods that are available in the Analysis++ package like ID3 and CobWeb. As classic algorithms require a fix structured input data set, it may be useful to create a input filter using a SOM and then to analyze this data with one of the previously developed methods. Also multi-layer SOMs can enhance accuracy, especially for clustering purpose, when disconnected clusters occur. While such an combined analysis is usually quite complex, as each routine is implemented in its own way or even in a completely different environment, it is extremely easy within the given package, as Input- Output data can be shared and reused.

7.1.2 New components

The flexibility of the framework concerns not only the implementation of new algorithms, but also the modification of existing ones. This can be done, by deriving new methods that replace previously implemented ones. The most important methods in this context are the neighborhood function and the adaption method (see section 4.2).

Initialization

As mentioned in section 2.1.2 several different approaches on setting up the initial net structure can be taken. Currently only a random initialization (within given bounds) is available `SOMInitialize`. This may be modified, to use as much knowledge about the input data as possible to create an according net structure. This will accelerate the approximation process and avoid the net from stopping in not optimal states. One may, for example, set up a grid, that is already unfolded (w.o. overlappings). Here statistical methods could be used to get a rough (and fast) approximation of the input data, while the net is used for detailed analysis.

Response

To find the *best matching node* the neighborhood function is used. By default, the Euclidean metric is used in `SOMBestMatching`, but any other metric is possible (see section 2.1.2 for the necessary metric properties). Changing the neighborhood function affects, how the net distributes over the input space. E.g. if the input data consists of parameters with a very different range, the Euclidean metric fails, as every dimension (parameter) is weighted the same. To prevent such abnormalities, one could use a distance measure, that normalizes each dimension over the input range.

Adaption

Updating the net offers a lot of possibilities to fit the algorithms to ones personal needs. Beside updating the adaption step `SOMAdaption` itself, several other classes may be varied. Using different neighborhood functions and learning rates allow to fine tune the net. At the current stage, only a one `SOMNeighborhoodFunction` and a two-step `SOMNeighborhoodFunctionTwoStep` neighborhood function (as used in Growing Neural Gas extension), are available. The learning rate `SOMLearningRate` is defined as linear decreasing. Other choices may be a logarithmic function or also a step function (see section 2.1.2).

7.1.3 Special Applications

To give an outlook, how Neural Maps can be used for special tasks, the application of Neural Maps for Clustering will be discussed in the following section.

As ANNs and especially SOMs are very good instruments for clustering, a lot of papers have been published. Nevertheless, only a few combine the abilities of growing neural maps with the clustering task.

If standard (growing) SOMs are used for clustering, the results are usable, but not always completely satisfying. This is illustrated in the Animal example, where the results are quite good, but in some cases strange results occur (Is a cat more like a duck then a tiger? See figure 6.5). They need a lot of (manual) fine-tuning, a lot of samples and often the output is hard to read (What are the clusters in the Animals sample?). These problems occur, because SOMs were designed for a more general usage and clustering is only a special application among others. If one wants to use the great flexibility and the structures generated by SOMs, but doesn't want to miss high-quality results, some modifications are necessary.

One modification may be, to remove "long" edges, as they connect very distinct parts of input data. This leads to separated clusters and eases interpretation of the results. This is somehow similar to the approach taken in the Growing Cell Structure algorithm, where nodes laying in regions with very few input samples are removed (see section 2.4.1). This may be done in a new `SOMAdaptionNet` step, where not only new nodes are added, but also removed if some conditions are satisfied. A simple variant of this approach is to display the maps edges in different colors. Poverty maps are another possibility to display distances in an two dimensional diagram [29, 30]. As the presentation of the results is not covered from the framework, external tools may be used to create such maps.

For clustering, supervised variants of SOMs are very interesting. Here a brought field of modifications are imaginable. A modified distance function could be used to learn the clusters while training. One could also use a error function that takes miss-classifications into account [23]. For that purpose, the average classification error is used instead of the average prediction error which is (usually) computed in the `SOMAdaption` class. This error measure can then be used within the `SOMAdaptionNet` method to locate that part of the map, where new neurons should be inserted. These leads to a high resolution in regions with very complex class borders, and only a few nodes in the center of each cluster.

7.2 Conclusion

In this paper a very interesting branch of Artificial Neural Nets, Growing Neural Maps has been described. Different algorithms were presented and analyzed to find a general framework. As an part of the Analysis++ project, the general framework for SOMs and some special methods were implemented in C++. This framework can be used as a basis for future development, as it allows fast and easy setup of new algorithms. Three extensions were presented to show this ability. At the end, three sample data-sets are analyzed, using these methods.

Building upon the basic structure developed in this paper, a broad range of future development is possible. Especially the development of new, more specific algorithms (e.g. for clustering) is provided.

This work can not provide a final solution, but furthermore open a door for future development, and I hope, this suggestion will be accepted.

Appendix A

Input files

The input is split up in two parts. The first part, stored in the `MapClass.inp` file, contains all strutural information like used classes and parameter descriptions. The second part, stored in the `MapInstances.inp` file, contains the data like nodes, edges and parameters for the operators. Also the *Instances* are stored here.

Input parameters can be given in to ways: As new parameters, given with name and type in the Class file and as value in the Input file, or as default parameters with trigger-name and value in the input file.

A.1 Map data

```
% create new map datatype
% derived from SOMMap
DataType: SOMMap new SOM Map
MyMap                           % name

% data classes
MyNode                          % node class
MyEdge                          % edge class

% operator classes
SOMInitializeRandom    % initialize class

SOMGenerateGrid        % generate class

SOMNeighborhoodFunction  % neighborhood function class
ClassNamePairs:        % parameters
END

SOMTrainNet            % trainnet class
ClassNamePairs:        % parameters
END

% plus TrainNet parameters
```

Figure A.1: SOMMap, MapClass.inp

```
Simple:                 % simple or detailed

NodeCount: 100          % inital value for numbering nodes
ClassNamePairs:         % additional node inforamtion
END
% node name
Node001
% wData (class Vector)
2                       % dimension of the vector
5 5                     % values
% node name
Node002
% wData (class Vector)
2                       % dimension of the vector
5 0                     % values
% node name
Node003
% wData (class Vector)
2                       % dimension of the vector
0 5                     % values
END
EdgeCount: 100          % initial value for
                        % numbering edges
ClassNamePairs:         % additional edge information
END
% edge name
Edge1
Node1 Node2             % connection nodes
END

LowerB: 2 0 0           % lower bound for the input data
UpperB: 2 5 5           % upper bound for the input data

% operator parameters:
SOMNeighborhoodFunctionParams:
% neighborhood function params
NoPreProcessNeighborhoodFunction  % do pre-processing
SOMTrainNetParams:
% train net parameters
NoPreProcessTrainNet              % do pre-processing

% plus TrainNet parameters
```

Figure A.2: SOMMap, MapInstances.inp

```
% create new node datatype
% derived from SOMMapNode
DataType: SOMMapNode node class
MyNode                            % name
MyWeight                          % weight class
```

Figure A.3: SOMMapNode, MapClass.inp

```
% node name
Node001
% wData (class Vector)
2         % dimension of the vector
5 5       % values
```

Figure A.4: SOMMapNode, MapInstances.inp

```
% create new weight datatype
% derived from SOMWeight
DataType: SOMWeight weight class
MyWeight                          % name
Vector                            % wData class
```

Figure A.5: SOMWeight, MapClass.inp

```
% wData (class Vector)
2         % dimension of the vector
5 5       % values
```

Figure A.6: SOMWeight, MapInstances.inp

A.2 Map operators

```
SOMTrainNet
ClassNamePairs:
t_max           Int       % maximum number of iterations

SOMAdaption               % adaption class
% Parameters

SOMBreakCondition         % break condition class
% Parameters

SOMGetNewSample           % get new sample class

END
```

Figure A.7: SOMTrainNet, MapClass.inp

```
SOMTrainNetParams
1000                      % t_max
NoPreProcessTrainNet

% plus additional parameters
```

Figure A.8: SOMTrainNet, MapInstances.inp

```
SOMBreakCondition
ClassNamePairs:
END
```

Figure A.9: SOMBreakCondition, MapClass.inp

```
SOMBreakConditionParams
% no params needed
```

Figure A.10: SOMBreakCondition, MapInstances.inp

```
SOMAdaption
ClassNamePairs:
END

SOMBestMatching          % BestMatching class
SOMUpdateNode            % UpdateNode class
SOMLearningRate          % LearningRate class
% Parameters
```

Figure A.11: SOMAdaption, MapClass.inp

```
SOMAdaptionParams
% no params needed
NoPreProcessAdaption

NoPreProcessUpdateNode
SOMLearningRateParams:
% learning rate params
```

Figure A.12: SOMAdaption, MapInstances.inp

```
SOMUpdateNode            % UpdateNode class
SOMLearningRate          % LearningRate class
```

Figure A.13: SOMUpdateNode, MapClass.inp

```
NoPreProcessUpdateNode
SOMLearningRateParams:
% learning rate params
```

Figure A.14: SOMUpdateNode, MapInstances.inp

```
SOMLearningRate
ClassNamePairs:
alpha_start     Real     % initial value for alpha
alpha_end       Real     % final   value for alpha
END
```

Figure A.15: SOMLearningRate, MapClass.inp

```
SOMLearningRateParams
0.6                        % alpha_start
0.1                        % alpha_end
```

Figure A.16: SOMLearningRate, MapInstances.inp

```
SOMNeighborhoodFunction
ClassNamePairs:
kwidth          Real       % initial width of the kernel
END
```

Figure A.17: SOMNeighborhoodFunction, MapClass.inp

```
SOMNeighborhoodFunctionParams
5                          % kwidth
NoPreProcessNeighborhoodFunction
```

Figure A.18: SOMNeighborhoodFunction, MapInstances.inp

A.3 Basic utilities

```
% create new edge datatype
% derived from SOMGrid
DataType: SOMGrid My new SOM Grid
MyMap                           % name
% data classes
MyNode                          % node class
MyEdge                          % edge class

% operator classes

% operator classes
SOMInitializeRandom    % initialize class

SOMGenerateGrid        % generate class

SOMNeighborhoodFunction  % neighborhood function class
ClassNamePairs:        % parameters
END

SOMTrainNet            % trainnet class
ClassNamePairs:        % parameters
END

% plus TrainNet parameters
```

Figure A.19: SOMGrid, MapClass.inp

```
Simple:                % simple or detailed

NodeCount: 100         % inital value for numbering nodes
ClassNamePairs:        % additional node inforamtion
END
% no nodes read in
END
EdgeCount: 100         % initial value for
                       % numbering edges
ClassNamePairs:        % additional edge information
END
% no edges read in
END

LowerB: 2 0 0          % lower bound for the input data
UpperB: 2 5 5          % upper bound for the input data

% operator parameters:
SOMNeighborhoodFunctionParams:
% neighborhood function params
NoPreProcessNeighborhoodFunction  % do pre-processing
SOMTrainNetParams:
% train net parameters

% dimension of the grid
Dim: 2 5 5
```

Figure A.20: SOMGrid, MapInstances.inp

```
% create new weight datatype
% derived from SOMWeight
DataType: SOMNumericVectorWeight weight class
MyWeight                          % name
Vector                            % wData class
```

Figure A.21: SOMNumericVectorWeight, MapClass.inp

```
% wData (class Vector)
2         % dimension of the vector
5 5       % values
```

Figure A.22: SOMNumericVectorWeight, MapInstances.inp

```
SOMGenerateNodes       % generate class
```

Figure A.23: SOMGenerateNodes, MapClass.inp

```
% <snip>
LowerB: 2 0 0          % lower bound for the input data
UpperB: 2 5 5          % upper bound for the input data

NewNodeCount:          % nodes to generate (without edge)

% operator parameters:
% <snip>
```

Figure A.24: SOMGenerateNodes, MapInstances.inp

```
SOMNeighborhoodFunctionTwoStep  % neighborhood function class
ClassNamePairs:
alpha_winning   Real            % alpha for the winning node
alpha_neighbors Real            % alpha for its neighbors
END
```

Figure A.25: SOMNeighborhoodFunctionTwoStep, MapClass.inp

```
SOMNeighborhoodFunctionParams:
0.05                            % alpha_winning
0.0006                          % alpha_neighbors
NoPreProcessNeighborhoodFunction
```

Figure A.26: SOMNeighborhoodFunctionTwoStep, MapInstances.inp

A.4 Neural Gas

```
SOMAdaptionNeuralGas
ClassNamePairs:
age_max_start   Int       % initial value for age_max
age_max_end     Int       % final   value for age_max
END
```

Figure A.27: SOMAdaptionNeuralGas, MapClass.inp

```
SOMAdaptionParams
10                        % age_max_start
100                       % age_max_end
NoPreProcessAdaption
```

Figure A.28: SOMAdaptionNeuralGas, MapInstances.inp

```
SOMNeighborhoodFunctionNeuralGas
ClassNamePairs:
lambda_start    Real      % initial value for lambda
lambda_end      Real      % final   value for lambda
END
```

Figure A.29: SOMNeighborhoodFunctionNeuralGas, MapClass.inp

```
SOMNeighborhoodFunctionParams
5.0                       % lambda_start
0.5                       % lambda_end
NoPreProcessNeighborhoodFunction
```

Figure A.30: SOMNeighborhoodFunctionNeuralGas, MapInstances.inp

A.5 Growing Neural Gas

```
SOMTrainNetGrowing
ClassNamePairs:
t_max           Integer   % maximum number of iterations
t_net           Integer   % number of iterations between
                          % a net adaption step
END

SOMAdaptionGNG            % adaption class
% Parameters

SOMBreakConditionGNG      % break condition class
% Parameters

SOMGetNewSample           % get new sample class

SOMAdaptionNetGNG         % AdaptionNet class
% Parameters
```

Figure A.31: SOMTrainNetGrowing, MapClass.inp

```
SOMTrainNetParams:
10000                     % t_max
300                       % t_net
NoPreProcessTrainNet

% plus additional parameters
```

Figure A.32: SOMTrainNetGrowing, MapInstances.inp

```
SOMBreakConditionGNG
ClassNamePairs:
maxNodes        Integer   % maximum number of nodes
END
```

Figure A.33: SOMBreakConditionGNG, MapClass.inp

```
SOMBreakConditionParams:
10                         % maxNodes
```

Figure A.34: SOMBreakConditionGNG, MapInstances.inp

```
SOMAdaptionGNG
ClassNamePairs:
age_max_start   Integer    % initial value for age_max
age_max_end     Integer    % final value for age_max
beta            Real       % parameter for error adaption
END

SOMBestMatchingNeuralGas  % BestMatching class

SOMUpdateNode              % UpdateNode class
% Parameters
```

Figure A.35: SOMAdaptionGNG, MapClass.inp

```
SOMAdaptionParams:
111                         % age_max_start
111                         % age_max_end
0.0005                      % beta
NoPreProcessAdaption
```

Figure A.36: SOMAdaptionGNG, MapInstances.inp

```
SOMAdaptionNetGNG
ClassNamePairs:
epsilon         Real       % parameter for error adaption
END
```

Figure A.37: SOMAdaptionNetGNG, MapClass.inp

```
SOMAdaptionNetParams:
0.5                        % epsilon
```

Figure A.38: SOMAdaptionNetGNG, MapInstances.inp

Appendix B

Test files

In this section the used input files are described. For detailed informations see chapter 6. For each sample, the Class and the Instance file is given. If special sample data was used, it is included in this files.

B.1 A simple sample

B.1.1 Kohonen Grid

Simple Sample Grid, GridClass.inp

```
% GridClass.inp
% a standard grid sample

ObjectClasses:

% create new map datatype
% derived from SOMNumericVectorWeight
DataType: SOMNumericVectorWeight my weight class
MyWeightClass                      % name
Vector                             % weight data class

% create new node datatype
% derived from SOMMapNode
DataType: SOMMapNode my node class
MyNodeClass                        % name
MyWeightClass                      % weight class

% create new map datatype
% derived from SOMMap
DataType: SOMGrid My new SOM Map
MyMapClass                         % name

% data classes
```

```
MyNodeClass                      % node class
Edge                             % edge class

% operator classes
SOMInitializeRandom              % initialize class

SOMGenerateGrid                  % generate class

SOMNeighborhoodFunction          % neighborhood f. class
ClassNamePairs:                  % parameters
kwidth          Real             % kernel width
END

SOMTrainNet                      % trainnet class
ClassNamePairs:                  % parameters
t_max           Integer          % maximum iterations
END

SOMAdaption                      % adaption class
ClassNamePairs:                  % parameters
END
SOMBestMatching                  % best matching class
SOMUpdateNode                    % UpdateNode class

SOMLearningRate                  % learning rate class
ClassNamePairs:                  % parameters
alpha_start     Real             % initial value for alpha
alpha_end       Real             % final value for alpha
END

SOMBreakCondition                % break condition class
ClassNamePairs:                  % parameters
END

SOMGetNewSample                  % get new sample class
END

% classes to read in
ClassNamePairs:
MyMapObject     MyMapClass
SOMParameters   KeyWords
SOMParamsOut    KeyWords
SOMInstances    KeyWords
END

% Instances
ObjectClasses:
END
```

```
ClassNamePairs:
END
```

Simple Sample Grid, GridInstances.inp

```
Sample --------------- Sample Graph Input --------------
Attributes:

%% MyMapObject     MyMapClass

% Read in ClassNamePairs only once for nodes and edges
Simple:
NodeCount: 100         % inital value for numbering nodes
ClassNamePairs:        % additional node inforamtion
END
% nodes:
% no nodes
END

EdgeCount: 100         % initial value for numbering edges
ClassNamePairs:        % additional edge information
END
% edges:
% no edges
END

LowerB: 2 0 0          % lower bound for the input data
UpperB: 2 5 5          % upper bound for the input data

% operator parameters:
SOMNeighborhoodFunctionParams:
3                      % kwidth
NoPreProcessNeighborhoodFunction  % do pre-processing?

SOMTrainNetParams:     % train net parameters
2000                   % t_max
PreProcessTrainNet   % do pre-processing?

SOMAdaptionParams:
% no parameters
NoPreProcessAdaption   % do pre-processing?

NoPreProcessUpdateNode % do pre-processing?

SOMLearningRateParams:
0.6                    % alpha_start
0.05                   % alpha_end
```

```
SOMBreakConditionParams:
% no parameters

Dim: 2 5 5            % 5x5 grid

%% SOMParameters   KeyWords
END

%% SOMParamsOut    KeyWords
END

%% SOMInstances    KeyWords
END

%% Instances:
END
```

B.1.2 Neural Gas

Simple Sample Neural Gas, NeuralGasClass.inp

```
% NeuralGasClass.inp
% a standard neural gas sample

ObjectClasses:

% create new map datatype
% derived from SOMNumericVectorWeight
DataType: SOMNumericVectorWeight my weight class
MyWeightClass                      % name
Vector                             % weight data class

% create new node datatype
% derived from SOMMapNode
DataType: SOMMapNode my node class
MyNodeClass                        % name
MyWeightClass                      % weight class

% create new map datatype
% derived from SOMMap
DataType: SOMMap My new SOM Map
MyMapClass                         % name

% data classes
MyNodeClass                        % node class
Edge                               % edge class

% operator classes
SOMInitializeRandom                % initialize class

SOMGenerateNodes                   % generate class

SOMNeighborhoodFunctionNeuralGas   % neighborhood f. class
ClassNamePairs:                    % parameters
lambda_start    Real               % inital value for lambda
lambda_end      Real               % final value for lambda
END

SOMTrainNet                        % trainnet class
ClassNamePairs:                    % parameters
t_max           Integer            % maximum iterations
END

SOMAdaptionNeuralGas               % adaption class
ClassNamePairs:                    % parameters
age_max_start   Integer            % initial age to remove edges
age_max_end     Integer            % final age to remove edges
```

```
END
SOMBestMatchingNeuralGas          % best matching class
SOMUpdateNode                     % UpdateNode class

SOMLearningRate                   % learning rate class
ClassNamePairs:                   % parameters
alpha_start     Real              % initial value for alpha
alpha_end       Real              % final value for alpha
END

SOMBreakCondition                 % break condition class
ClassNamePairs:                   % parameters
END

SOMGetNewSample                   % get new sample class
END

% classes to read in
ClassNamePairs:
MyMapObject     MyMapClass
SOMParameters   KeyWords
SOMParamsOut    KeyWords
SOMInstances    KeyWords
END

% Instances
ObjectClasses:
END

ClassNamePairs:
END
```

Simple Sample Neural Gas, NeuralGasInstances.inp

```
Sample ------------ Sample Neural Gas Input -----------
Attributes:

%% MyMapObject     MyMapClass

% Read in ClassNamePairs only once for nodes and edges
Simple:
NodeCount: 100         % inital value for numbering nodes
ClassNamePairs:        % additional node inforamtion
END
% nodes:
% no nodes
END
```

```
EdgeCount: 100          % initial value for numbering edges
ClassNamePairs:         % additional edge information
END
% edges:
% no edges
END

LowerB: 2 0 0           % lower bound for the input data
UpperB: 2 5 5           % upper bound for the input data

% operator parameters:
% GenerateNodes
NewNodeCount: 20        % number of initial nodes

SOMNeighborhoodFunctionParams:
5.0                     % lambda_start
0.01                    % lambda_end
NoPreProcessNeighborhoodFunction  % do pre-processing?

SOMTrainNetParams:      % train net parameters
4000                    % t_max
NoPreProcessTrainNet    % do pre-processing?

SOMAdaptionParams:
20                      % age_max_start
200                     % age_max_end
NoPreProcessAdaption    % do pre-processing?

NoPreProcessUpdateNode % do pre-processing?

SOMLearningRateParams:
0.5                     % alpha_start
0.005                   % alpha_end

SOMBreakConditionParams:
% no parameters

%% SOMParameters   KeyWords
END

%% SOMParamsOut    KeyWords
END

%% SOMInstances    KeyWords
END

%% Instances:
END
```

B.1.3 Growing Neural Gas

Simple Sample GNG, GNGClass.inp

```
% GNGClass.inp
% a standard growing neural gas sample

ObjectClasses:

% create new map datatype
% derived from SOMNumericVectorWeight
DataType: SOMNumericVectorWeight my weight class
MyWeightClass                     % name
Vector                            % weight data class

% create new node datatype
% derived from SOMMapNode
DataType: SOMMapNode my node class
MyNodeClass                       % name
MyWeightClass                     % weight class

% create new map datatype
% derived from SOMMap
DataType: SOMMap My new SOM Map
MyMapClass                        % name

% data classes
MyNodeClass                       % node class
Edge                              % edge class

% operator classes
SOMInitializeRandom               % initialize class

SOMGenerateNodes                  % generate class

SOMNeighborhoodFunctionTwoStep    % neighborhood f. class
ClassNamePairs:                   % parameters
alpha_winning   Real              % alpha for winning node
alpha_neighbors Real              % alpha for neighbored nodes
END

SOMTrainNetGrowing                % trainnet class
ClassNamePairs:                   % parameters
t_max           Integer           % maximum iterations
t_net           Integer           % iterations between net adaptions
END

SOMAdaptionGNG                    % adaption class
ClassNamePairs:                   % parameters
age_max_start   Integer           % initial age to remove edges
```

```
age_max_end     Integer           % final age to remove edges
beta            Real              %
END
SOMBestMatchingNeuralGas          % best matching class
SOMUpdateNode                     % UpdateNode class

SOMLearningRate                   % learning rate class
ClassNamePairs:                   % parameters
alpha_start     Real              % initial value for alpha
alpha_end       Real              % final value for alpha
END

SOMBreakConditionGNG              % break condition class
ClassNamePairs:                   % parameters
maxNodes        Integer           % maximum number of nodes
END

SOMGetNewSample                   % get new sample class

SOMAdaptionNetGNG                 % adaption net class
ClassNamePairs:                   % parameters
epsilon         Real              %
END
END

% classes to read in
ClassNamePairs:
MyMapObject     MyMapClass
SOMParameters   KeyWords
SOMParamsOut    KeyWords
SOMInstances    KeyWords
END

% Instances
ObjectClasses:
END

ClassNamePairs:
END
```

Simple Sample GNG, GNGInstances.inp

```
Sample -------- Sample Growing Neural Gas Input -------
Attributes:

%% MyMapObject     MyMapClass

% Read in ClassNamePairs only once for nodes and edges
```

```
Simple:
NodeCount: 100         % inital value for numbering nodes
ClassNamePairs:        % additional node inforamtion
END
% nodes:
% no nodes
END

EdgeCount: 100         % initial value for numbering edges
ClassNamePairs:        % additional edge information
END
% edges:
% no edges
END

LowerB: 2 0 0          % lower bound for the input data
UpperB: 2 5 5          % upper bound for the input data

NewNodeCount: 2        % number of initial nodes

SOMNeighborhoodFunctionParams:
0.05                   % alpha_winning
0.005                  % alpha_neighbors
NoPreProcessNeighborhoodFunction  % do pre-processing?

SOMTrainNetParams:     % train net parameters
10000                  % t_max
400                    % t_net
NoPreProcessTrainNet   % do pre-processing?

SOMAdaptionParams:
311                    % age_max_start
311                    % age_max_end
0.005                  % beta
NoPreProcessAdaption   % do pre-processing?

NoPreProcessUpdateNode % do pre-processing?

SOMLearningRateParams:
1                      % alpha_start
0.5                    % alpha_end

SOMBreakConditionParams:
20                     % maxNodes

SOMAdaptionNetParams:
0.5                    % epsilon

%% SOMParameters   KeyWords
```

```
END

%% SOMParamsOut    KeyWords
END

%% SOMInstances    KeyWords
END

%% Instances:
END
```

B.2 Animals

B.2.1 Animals Data set

Animals Data Set, AnimalsClass.inp

```
% AnimalsClass.inp
% animal example

ObjectClasses:
END

% classes to read in
ClassNamePairs:
AttributeNames          KeyWords
GoalAttributes          KeyWords
TrainInstanceNameList   KeyWords
TestInstanceNameList    KeyWords
END

% Instances
ObjectClasses:
END

ClassNamePairs:
small     Real
medium    Real
big       Real

2legs     Real
4legs     Real
hair      Real
hooves    Real
mane      Real
feathers  Real

hunt      Real
run       Real
fly       Real
swim      Real

class     Real
END
```

Animals Data Set, AnimalsInstances.inp

```
Animals --------------- Animals Instances --------------
Attributes:
```

```
%% AttributeNames   KeyWords
small medium big 2legs 4legs hair hooves
mane feathers hunt run fly swim
END

%% GoalAttributes    KeyWords
class
END

%% TrainInstanceNameList    KeyWords
dove hen duck goose owl hawk eagle fox dog wolf
cat tiger lion horse zebra cow
END

%% TestInstanceNameList    KeyWords
dove hen duck goose owl hawk eagle fox dog wolf
cat tiger lion horse zebra cow
END

%% Instances:
Instance: 1 dove animal
1 0 0  1 0 0 0 0 1  0 0 1 0  1
Instance: 1 hen animal
1 0 0  1 0 0 0 0 1  0 0 0 0  1
Instance: 1 duck animal
1 0 0  1 0 0 0 0 1  0 0 1 1  1
Instance: 1 goose animal
1 0 0  1 0 0 0 0 1  0 0 1 1  1
Instance: 1 owl animal
1 0 0  1 0 0 0 0 1  1 0 1 0  1
Instance: 1 hawk animal
1 0 0  1 0 0 0 0 1  1 0 1 0  1
Instance: 1 eagle animal
0 1 0  1 0 0 0 0 1  1 0 0 0  1
Instance: 1 fox animal
0 1 0  0 1 1 0 0 0  1 0 0 0  2
Instance: 1 dog animal
0 1 0  0 1 1 0 0 0  0 1 0 0  2
Instance: 1 wolf animal
0 1 0  0 1 1 0 1 0  1 1 0 0  2
Instance: 1 cat animal
1 0 0  0 1 1 0 0 0  1 0 0 0  2
Instance: 1 tiger animal
0 0 1  0 1 1 0 0 0  1 1 0 0  2
Instance: 1 lion animal
0 0 1  0 1 1 0 1 0  1 1 0 0  2
Instance: 1 horse animal
0 0 1  0 1 1 1 1 0  0 1 0 0  3
Instance: 1 zebra animal
```

```
0 0 1  0 1 1 1 1 0  0 1 0 0  3
Instance: 1 cow animal
0 0 1  0 1 1 1 0 0  0 0 0 0  3
END
```

B.2.2 Kohonen Grid

Animals Grid, AnimalsGridClass.inp

```
% AnimalsGridClass.inp
% animal example using a grid

ObjectClasses:

% create new map datatype
% derived from SOMNumericVectorWeight
DataType: SOMNumericVectorWeight my weight class
MyWeightClass                      % name
Vector                             % weight data class

% create new node datatype
% derived from SOMMapNode
DataType: SOMMapNode my node class
MyNodeClass                        % name
MyWeightClass                      % weight class

% create new map datatype
% derived from SOMMap
DataType: SOMGrid My new SOM Map
MyMapClass                         % name

% data classes
MyNodeClass                        % node class
Edge                               % edge class

% operator classes
SOMInitializeRandom                % initialize class

SOMGenerateGrid                    % generate class

SOMNeighborhoodFunction            % neighborhood f. class
ClassNamePairs:                    % parameters
kwidth          Real               % kernel width
END

SOMTrainNet                        % trainnet class
ClassNamePairs:                    % parameters
t_max           Integer            % maximum iterations
END

SOMAdaption                        % adaption class
ClassNamePairs:                    % parameters
END
SOMBestMatching                    % best matching class
SOMUpdateNode                      % UpdateNode class
```

```
SOMLearningRate                  % learning rate class
ClassNamePairs:                  % parameters
alpha_start     Real             % initial value for alpha
alpha_end       Real             % final value for alpha
END

SOMBreakCondition                % break condition class
ClassNamePairs:                  % parameters
END

SOMGetNewSampleInstance          % get new sample class
END

% classes to read in
ClassNamePairs:
SOMMap                   MyMapClass
END

% Instances
ObjectClasses:
END

ClassNamePairs:
END
```

Animals Grid, AnimalsGridInstances.inp

```
Animals --------------- Animals Grid Input --------------
Attributes:

%% MyMapObject     MyMapClass

% Read in ClassNamePairs only once for nodes and edges
Simple:
NodeCount: 100        % inital value for numbering nodes
ClassNamePairs:       % additional node inforamtion
END
% nodes:
% no nodes
END

EdgeCount: 100        % initial value for numbering edges
ClassNamePairs:       % additional edge information
END
% edges:
% no edges
END
```

```
% lower bound (always 0)
LowerB: 13 0 0 0 0 0 0 0 0 0 0 0 0 0
% upper bound (always 1)
UpperB: 13 1 1 1 1 1 1 1 1 1 1 1 1 1

% operator parameters:
SOMNeighborhoodFunctionParams:
2                      % kwidth
NoPreProcessNeighborhoodFunction  % do pre-processing?

SOMTrainNetParams:     % train net parameters
1000                   % t_max
PreProcessTrainNet     % do pre-processing?

SOMAdaptionParams:
% no parameters
NoPreProcessAdaption   % do pre-processing?

NoPreProcessUpdateNode % do pre-processing?

SOMLearningRateParams:
0.5                    % alpha_start
0.005                  % alpha_end

SOMBreakConditionParams:
% no parameters

Dim: 2 5 5             % 5x5 grid
END
```

B.2.3 Neural Gas

Animals Neural Gas, AnimalsNGClass.inp

```
% AnimalsNGClass.inp
% animals example using a NeuralGass

ObjectClasses:

% create new map datatype
% derived from SOMNumericVectorWeight
DataType: SOMNumericVectorWeight my weight class
MyWeightClass                      % name
Vector                             % weight data class

% create new node datatype
% derived from SOMMapNode
DataType: SOMMapNode my node class
MyNodeClass                        % name
MyWeightClass                      % weight class

% create new map datatype
% derived from SOMMap
DataType: SOMMap My new SOM Map
MyMapClass                         % name

% data classes
MyNodeClass                        % node class
Edge                               % edge class

% operator classes
SOMInitializeRandom                % initialize class

SOMGenerateNodes                   % generate class

SOMNeighborhoodFunctionNeuralGas   % neighborhood f. class
ClassNamePairs:                    % parameters
lambda_start    Real               % inital value for lambda
lambda_end      Real               % final value for lambda
END

SOMTrainNet                        % trainnet class
ClassNamePairs:                    % parameters
t_max           Integer            % maximum iterations
END

SOMAdaptionNeuralGas               % adaption class
ClassNamePairs:                    % parameters
age_max_start   Integer            % initial age to remove edges
age_max_end     Integer            % final age to remove edges
```

```
END
SOMBestMatchingNeuralGas         % best matching class
SOMUpdateNode                    % UpdateNode class

SOMLearningRate                  % learning rate class
ClassNamePairs:                  % parameters
alpha_start     Real             % initial value for alpha
alpha_end       Real             % final value for alpha
END

SOMBreakCondition                % break condition class
ClassNamePairs:                  % parameters
END

SOMGetNewSampleInstance          % get new sample class
END

% classes to read in
ClassNamePairs:
SOMMap     MyMapClass
END

% Instances
ObjectClasses:
END

ClassNamePairs:
END
```

Animals Neural Gas, AnimalsNGInstances.inp

```
Animals ------------ Animals Neural Gas Input -----------
Attributes:

%% MyMapObject     MyMapClass

% Read in ClassNamePairs only once for nodes and edges
Simple:
NodeCount: 100         % inital value for numbering nodes
ClassNamePairs:        % additional node inforamtion
END
% nodes:
% no nodes
END

EdgeCount: 100         % initial value for numbering edges
ClassNamePairs:        % additional edge information
END
```

```
% edges:
% no edges
END

% lower bound (always 0)
LowerB: 13 0 0 0 0 0 0 0 0 0 0 0 0 0
% upper bound (always 1)
UpperB: 13 1 1 1 1 1 1 1 1 1 1 1 1 1

% operator parameters:
% GenerateNodes
NewNodeCount: 16       % number of initial nodes

SOMNeighborhoodFunctionParams:
5.0                    % lambda_start
0.01                   % lambda_end
NoPreProcessNeighborhoodFunction  % do pre-processing?

SOMTrainNetParams:     % train net parameters
1000                   % t_max
NoPreProcessTrainNet   % do pre-processing?

SOMAdaptionParams:
10                     % age_max_start
100                    % age_max_end
NoPreProcessAdaption   % do pre-processing?

NoPreProcessUpdateNode % do pre-processing?

SOMLearningRateParams:
0.5                    % alpha_start
0.005                  % alpha_end

SOMBreakConditionParams:
% no parameters

END
```

B.3 Iris

B.3.1 Iris Data set

Iris Data Set, IrisClass.inp

```
% IrisClass.inp
% iris example general setup

ObjectClasses:
END

% classes to read in
ClassNamePairs:
AttributeNames          KeyWords
GoalAttributes          KeyWords
TrainInstanceNameList   KeyWords
TestInstanceNameList    KeyWords
END

% Instances
ObjectClasses:
DataType:
NValued The iris type
IrisType Iris-versicolor 1.0 Iris-setosa 0.5 Iris-virginica 0.0 END
END

ClassNamePairs:
sepal_length    Real
sepal_width     Real
petal_length    Real
petal_width     Real
petal_type      IrisType
END
```

Iris Data Set, IrisInstances.inp

```
% IrisInstances.inp
% iris example general setup

Attributes:
%% AttributeNames  KeyWords
sepal_length sepal_width petal_length petal_width
END

%% GoalAttributes   KeyWords
petal_type
END
```

```
%% TrainInstanceNameList    KeyWords
Flower1 Flower2 Flower3 Flower4 Flower5
Flower6 Flower7 Flower8 Flower9 Flower10
Flower11 Flower12 Flower13 Flower14 Flower15
Flower16 Flower17 Flower18 Flower19 Flower20
Flower21 Flower22 Flower23 Flower24 Flower25
Flower26 Flower27 Flower28 Flower29 Flower30
Flower31 Flower32 Flower33 Flower34 Flower35
Flower36 Flower37 Flower38 Flower39 Flower40
Flower41 Flower42 Flower43 Flower44 Flower45
Flower46 Flower47 Flower48 Flower49 Flower50
Flower51 Flower52 Flower53 Flower54 Flower55
Flower56 Flower57 Flower58 Flower59 Flower60
Flower61 Flower62 Flower63 Flower64 Flower65
Flower66 Flower67 Flower68 Flower69 Flower70
Flower71 Flower72 Flower73 Flower74 Flower75
Flower76 Flower77 Flower78 Flower79 Flower80
Flower81 Flower82 Flower83 Flower84 Flower85
Flower86 Flower87 Flower88 Flower89 Flower90
Flower91 Flower92 Flower93 Flower94 Flower95
Flower96 Flower97 Flower98 Flower99 Flower100
END

%% TestInstanceNameList    KeyWords
Flower1 Flower2 Flower3 Flower4 Flower5
Flower6 Flower7 Flower8 Flower9 Flower10
Flower11 Flower12 Flower13 Flower14 Flower15
Flower16 Flower17 Flower18 Flower19 Flower20
Flower21 Flower22 Flower23 Flower24 Flower25
Flower26 Flower27 Flower28 Flower29 Flower30
Flower31 Flower32 Flower33 Flower34 Flower35
Flower36 Flower37 Flower38 Flower39 Flower40
Flower41 Flower42 Flower43 Flower44 Flower45
Flower46 Flower47 Flower48 Flower49 Flower50
Flower51 Flower52 Flower53 Flower54 Flower55
Flower56 Flower57 Flower58 Flower59 Flower60
Flower61 Flower62 Flower63 Flower64 Flower65
Flower66 Flower67 Flower68 Flower69 Flower70
Flower71 Flower72 Flower73 Flower74 Flower75
Flower76 Flower77 Flower78 Flower79 Flower80
Flower81 Flower82 Flower83 Flower84 Flower85
Flower86 Flower87 Flower88 Flower89 Flower90
Flower91 Flower92 Flower93 Flower94 Flower95
Flower96 Flower97 Flower98 Flower99 Flower100
END

%% Instances:
Instance: 1 Flower1 The 1st Flower
```

```
6.3     3.3     4.7     1.6     Iris-versicolor
Instance: 2 Flower2 The 2nd Flower
5.8     4       1.2     0.2     Iris-setosa
Instance: 3 Flower3 The 3rd Flower
6.7     3.3     5.7     2.1     Iris-virginica
Instance: 4 Flower4 The 4rd Flower
6.1     2.8     4       1.3     Iris-versicolor
Instance: 5 Flower5 The 5rd Flower
5       3.4     1.5     0.2     Iris-setosa
Instance: 6 Flower6 The 6rd Flower
6.2     2.2     4.5     1.5     Iris-versicolor
Instance: 7 Flower7 The 7rd Flower
7.2     3.6     6.1     2.5     Iris-virginica
Instance: 8 Flower8 The 8rd Flower
4.4     2.9     1.4     0.2     Iris-setosa
Instance: 9 Flower9 The 9rd Flower
5       3.5     1.3     0.3     Iris-setosa
Instance: 10 Flower10 The 10st Flower
5.4     3.4     1.5     0.4     Iris-setosa
Instance: 11 Flower11 The 11st Flower
6.3     2.8     5.1     1.5     Iris-virginica
Instance: 12 Flower12 The 12rd Flower
5.5     4.2     1.4     0.2     Iris-setosa
Instance: 13 Flower13 The 13rd Flower
5.5     2.6     4.4     1.2     Iris-versicolor
Instance: 14 Flower14 The 14rd Flower
5.5     2.4     3.7     1       Iris-versicolor
Instance: 15 Flower15 The 15rd Flower
7.7     3       6.1     2.3     Iris-virginica
Instance: 16 Flower16 The 16rd Flower
4.6     3.1     1.5     0.2     Iris-setosa
Instance: 17 Flower17 The 17rd Flower
6.8     2.8     4.8     1.4     Iris-versicolor
Instance: 18 Flower18 The 18rd Flower
5.8     2.7     4.1     1       Iris-versicolor
Instance: 19 Flower19 The 19rd Flower
4.7     3.2     1.6     0.2     Iris-setosa
Instance: 20 Flower20 The 20st Flower
6.5     3.2     5.1     2       Iris-virginica
Instance: 21 Flower21 The 21rd Flower
5.1     3.5     1.4     0.3     Iris-setosa
Instance: 22 Flower22 The 22rd Flower
6.3     2.7     4.9     1.8     Iris-virginica
Instance: 23 Flower23 The 23rd Flower
7.1     3       5.9     2.1     Iris-virginica
Instance: 24 Flower24 The 24rd Flower
6.3     2.5     4.9     1.5     Iris-versicolor
Instance: 25 Flower25 The 25rd Flower
6.1     3       4.9     1.8     Iris-virginica
```

```
Instance: 26 Flower26 The 26rd Flower
5.4     3.7     1.5     0.2     Iris-setosa
Instance: 27 Flower27 The 27rd Flower
5.4     3.9     1.3     0.4     Iris-setosa
Instance: 28 Flower28 The 28rd Flower
6.4     2.7     5.3     1.9     Iris-virginica
Instance: 29 Flower29 The 29rd Flower
6.9     3.2     5.7     2.3     Iris-virginica
Instance: 30 Flower30 The 30st Flower
5.1     3.8     1.9     0.4     Iris-setosa
Instance: 31 Flower31 The 31rd Flower
7.2     3.2     6       1.8     Iris-virginica
Instance: 32 Flower32 The 32rd Flower
6.3     2.3     4.4     1.3     Iris-versicolor
Instance: 33 Flower33 The 33rd Flower
6.6     3       4.4     1.4     Iris-versicolor
Instance: 34 Flower34 The 34rd Flower
6.3     3.3     6       2.5     Iris-virginica
Instance: 35 Flower35 The 35rd Flower
6.7     3.1     4.4     1.4     Iris-versicolor
Instance: 36 Flower36 The 36rd Flower
7.7     2.8     6.7     2       Iris-virginica
Instance: 37 Flower37 The 37rd Flower
5.7     3       4.2     1.2     Iris-versicolor
Instance: 38 Flower38 The 38rd Flower
6.9     3.1     4.9     1.5     Iris-versicolor
Instance: 39 Flower39 The 39rd Flower
6.5     3       5.5     1.8     Iris-virginica
Instance: 40 Flower40 The 40st Flower
6       2.7     5.1     1.6     Iris-versicolor
Instance: 41 Flower41 The 41rd Flower
4.8     3.4     1.9     0.2     Iris-setosa
Instance: 42 Flower42 The 42rd Flower
6.4     3.2     5.3     2.3     Iris-virginica
Instance: 43 Flower43 The 43rd Flower
6       2.9     4.5     1.5     Iris-versicolor
Instance: 44 Flower44 The 44rd Flower
4.9     2.4     3.3     1       Iris-versicolor
Instance: 45 Flower45 The 45rd Flower
6.6     2.9     4.6     1.3     Iris-versicolor
Instance: 46 Flower46 The 46rd Flower
7       3.2     4.7     1.4     Iris-versicolor
Instance: 47 Flower47 The 47rd Flower
6.3     2.5     5       1.9     Iris-virginica
Instance: 48 Flower48 The 48rd Flower
5       3       1.6     0.2     Iris-setosa
Instance: 49 Flower49 The 49rd Flower
6.7     3.3     5.7     2.5     Iris-virginica
Instance: 50 Flower50 The 50st Flower
```

```
5.8     2.6     4       1.2     Iris-versicolor
Instance: 51 Flower51 The 51rd Flower
6       3.4     4.5     1.6     Iris-versicolor
Instance: 52 Flower52 The 52rd Flower
5       3.6     1.4     0.2     Iris-setosa
Instance: 53 Flower53 The 53rd Flower
5       3.4     1.6     0.4     Iris-setosa
Instance: 54 Flower54 The 54rd Flower
6.9     3.1     5.4     2.1     Iris-virginica
Instance: 55 Flower55 The 55rd Flower
6.8     3.2     5.9     2.3     Iris-virginica
Instance: 56 Flower56 The 56rd Flower
6.9     3.1     5.1     2.3     Iris-virginica
Instance: 57 Flower57 The 57rd Flower
5.2     3.4     1.4     0.2     Iris-setosa
Instance: 58 Flower58 The 58rd Flower
5.5     2.3     4       1.3     Iris-versicolor
Instance: 59 Flower59 The 59rd Flower
6.4     2.9     4.3     1.3     Iris-versicolor
Instance: 60 Flower60 The 50st Flower
5.6     2.8     4.9     2       Iris-virginica
Instance: 61 Flower61 The 1rd Flower
5       2       3.5     1       Iris-versicolor
Instance: 62 Flower62 The 2rd Flower
5.1     3.3     1.7     0.5     Iris-setosa
Instance: 63 Flower63 The 3rd Flower
6.7     3       5.2     2.3     Iris-virginica
Instance: 64 Flower64 The 4rd Flower
4.6     3.2     1.4     0.2     Iris-setosa
Instance: 65 Flower65 The 5rd Flower
5.3     3.7     1.5     0.2     Iris-setosa
Instance: 66 Flower66 The 6rd Flower
6.1     2.6     5.6     1.4     Iris-virginica
Instance: 67 Flower67 The 7rd Flower
5.5     2.5     4       1.3     Iris-versicolor
Instance: 68 Flower68 The 8rd Flower
6.1     3       4.6     1.4     Iris-versicolor
Instance: 69 Flower69 The 9rd Flower
4.8     3       1.4     0.3     Iris-setosa
Instance: 70 Flower70 The 0st Flower
4.9     2.5     4.5     1.7     Iris-virginica
Instance: 71 Flower71 The 1rd Flower
7.9     3.8     6.4     2       Iris-virginica
Instance: 72 Flower72 The 2rd Flower
5.5     2.4     3.8     1.1     Iris-versicolor
Instance: 73 Flower73 The 3rd Flower
5.4     3.9     1.7     0.4     Iris-setosa
Instance: 74 Flower74 The 4rd Flower
5.9     3       4.2     1.5     Iris-versicolor
```

```
Instance: 75 Flower75 The 5rd Flower
4.8     3.1     1.6     0.2     Iris-setosa
Instance: 76 Flower76 The 6rd Flower
5.2     2.7     3.9     1.4     Iris-versicolor
Instance: 77 Flower77 The 7rd Flower
4.4     3.2     1.3     0.2     Iris-setosa
Instance: 78 Flower78 The 8rd Flower
6.2     3.4     5.4     2.3     Iris-virginica
Instance: 79 Flower79 The 9rd Flower
5.5     3.5     1.3     0.2     Iris-setosa
Instance: 80 Flower80 The 0st Flower
4.9     3.1     1.5     0.1     Iris-setosa
Instance: 81 Flower81 The 1rd Flower
4.7     3.2     1.3     0.2     Iris-setosa
Instance: 82 Flower82 The 2rd Flower
5       3.2     1.2     0.2     Iris-setosa
Instance: 83 Flower83 The 3rd Flower
5.6     2.9     3.6     1.3     Iris-versicolor
Instance: 84 Flower84 The 4rd Flower
7.3     2.9     6.3     1.8     Iris-virginica
Instance: 85 Flower85 The 5rd Flower
5.7     4.4     1.5     0.4     Iris-setosa
Instance: 86 Flower86 The 6rd Flower
4.5     2.3     1.3     0.3     Iris-setosa
Instance: 87 Flower87 The 7rd Flower
6.5     2.8     4.6     1.5     Iris-versicolor
Instance: 88 Flower88 The 8rd Flower
5.7     2.8     4.1     1.3     Iris-versicolor
Instance: 89 Flower89 The 9rd Flower
4.9     3.1     1.5     0.1     Iris-setosa
Instance: 90 Flower90 The 0rd Flower
6.1     2.9     4.7     1.4     Iris-versicolor
Instance: 91 Flower91 The 1rd Flower
4.6     3.4     1.4     0.3     Iris-setosa
Instance: 92 Flower92 The 2rd Flower
4.9     3.1     1.5     0.1     Iris-setosa
Instance: 93 Flower93 The 20rd Flower
5.7     3.8     1.7     0.3     Iris-setosa
Instance: 94 Flower94 The 30rd Flower
7.7     3.8     6.7     2.2     Iris-virginica
Instance: 95 Flower95 The 40rd Flower
7.2     3       5.8     1.6     Iris-virginica
Instance: 96 Flower96 The 50rd Flower
5.8     2.8     5.1     2.4     Iris-virginica
Instance: 97 Flower97 The 50rd Flower
6.7     2.5     5.8     1.8     Iris-virginica
Instance: 98 Flower98 The 0rd Flower
5.1     3.7     1.5     0.4     Iris-setosa
Instance: 99 Flower99 The 0rd Flower
```

```
6       2.2     4       1       Iris-versicolor
Instance: 100 Flower100 The 0st Flower
6.3     3.4     5.6     2.4     Iris-virginica
END
```

B.3.2 Neural Gas

Iris Neural Gas, IrisNGClass.inp

```
% IrisNGClass.inp
% iris example using a NeuralGas

ObjectClasses:

% create new map datatype
% derived from SOMNumericVectorWeight
DataType: SOMNumericVectorWeight my weight class
MyWeightClass                      % name
Vector                             % weight data class

% create new node datatype
% derived from SOMMapNode
DataType: SOMMapNode my node class
MyNodeClass                        % name
MyWeightClass                      % weight class

% create new map datatype
% derived from SOMMap
DataType: SOMMap My new SOM Map
MyMapClass                         % name

% data classes
MyNodeClass                        % node class
Edge                               % edge class

% operator classes
SOMInitializeRandom                % initialize class

SOMGenerateNodes                   % generate class

SOMNeighborhoodFunctionNeuralGas   % neighborhood f. class
ClassNamePairs:                    % parameters
lambda_start    Real               % inital value for lambda
lambda_end      Real               % final value for lambda
END

SOMTrainNet                        % trainnet class
ClassNamePairs:                    % parameters
t_max           Integer            % maximum iterations
END

SOMAdaptionNeuralGas               % adaption class
ClassNamePairs:                    % parameters
age_max_start   Integer            % initial age to remove edges
age_max_end     Integer            % final age to remove edges
```

```
END
SOMBestMatchingNeuralGas          % best matching class
SOMUpdateNode                     % UpdateNode class

SOMLearningRate                   % learning rate class
ClassNamePairs:                   % parameters
alpha_start     Real              % initial value for alpha
alpha_end       Real              % final value for alpha
END

SOMBreakCondition                 % break condition class
ClassNamePairs:                   % parameters
END

SOMGetNewSampleInstance           % get new sample class
END

% classes to read in
ClassNamePairs:
SOMMap     MyMapClass
END

% Instances
ObjectClasses:
END

ClassNamePairs:
END
```

Iris Neural Gas, IrisNGInstances.inp

```
Animals ------------ Animals Neural Gas Input -----------
Attributes:

%% MyMapObject     MyMapClass

% Read in ClassNamePairs only once for nodes and edges
Simple:
NodeCount: 100         % inital value for numbering nodes
ClassNamePairs:        % additional node inforamtion
END
% nodes:
% no nodes
END

EdgeCount: 100         % initial value for numbering edges
ClassNamePairs:        % additional edge information
END
```

```
% edges:
% no edges
END

% lower bound
LowerB: 4 5 2 0 0
% upper bound
UpperB: 4 8 4 7 3

% operator parameters:
% GenerateNodes
NewNodeCount: 10       % number of initial nodes

SOMNeighborhoodFunctionParams:
5.0                    % lambda_start
0.01                   % lambda_end
NoPreProcessNeighborhoodFunction  % do pre-processing?

SOMTrainNetParams:     % train net parameters
2000                   % t_max
NoPreProcessTrainNet   % do pre-processing?

SOMAdaptionParams:
10                     % age_max_start
100                    % age_max_end
NoPreProcessAdaption   % do pre-processing?

NoPreProcessUpdateNode % do pre-processing?

SOMLearningRateParams:
0.5                    % alpha_start
0.005                  % alpha_end

SOMBreakConditionParams:
% no parameters

END
```

B.3.3 Growing Neural Gas

Iris Growing Neural Gas, IrisGNGClass.inp

```
% IrisGNGClass.inp
% iris example using a growing neural gas

ObjectClasses:

% create new map datatype
% derived from SOMNumericVectorWeight
DataType: SOMNumericVectorWeight my weight class
MyWeightClass                      % name
Vector                             % weight data class

% create new node datatype
% derived from SOMMapNode
DataType: SOMMapNode my node class
MyNodeClass                        % name
MyWeightClass                      % weight class

% create new map datatype
% derived from SOMMap
DataType: SOMMap My new SOM Map
MyMapClass                         % name

% data classes
MyNodeClass                        % node class
Edge                               % edge class

% operator classes
SOMInitializeRandom                % initialize class

SOMGenerateNodes                   % generate class

SOMNeighborhoodFunctionTwoStep     % neighborhood f. class
ClassNamePairs:                    % parameters
alpha_winning   Real               % alpha for winning node
alpha_neighbors Real               % alpha for neighbored nodes
END

SOMTrainNetGrowing                 % trainnet class
ClassNamePairs:                    % parameters
t_max           Integer            % maximum iterations
t_net           Integer            % iterations between net adaptions
END

SOMAdaptionGNG                     % adaption class
ClassNamePairs:                    % parameters
age_max_start   Integer            % initial age to remove edges
```

```
age_max_end     Integer             % final age to remove edges
beta            Real                %
END
SOMBestMatchingNeuralGas            % best matching class
SOMUpdateNode                       % UpdateNode class

SOMLearningRate                     % learning rate class
ClassNamePairs:                     % parameters
alpha_start     Real                % initial value for alpha
alpha_end       Real                % final value for alpha
END

SOMBreakConditionGNG                % break condition class
ClassNamePairs:                     % parameters
maxNodes        Integer             % maximum number of nodes
END

SOMGetNewSampleInstance             % get new sample class

SOMAdaptionNetGNG                   % adaption net class
ClassNamePairs:                     % parameters
epsilon         Real                %
END
END

% classes to read in
ClassNamePairs:
SOMMap     MyMapClass
END

% Instances
ObjectClasses:
END

ClassNamePairs:
END
```

Iris Growing Neural Gas, IrisGNGInstances.inp

```
Sample -------- Iris Growing Neural Gas Input -------
Attributes:

%% MyMapObject     MyMapClass

% Read in ClassNamePairs only once for nodes and edges
Simple:
NodeCount: 100         % inital value for numbering nodes
ClassNamePairs:        % additional node inforamtion
```

```
END
% nodes:
% no nodes
END

EdgeCount: 100        % initial value for numbering edges
ClassNamePairs:       % additional edge information
END
% edges:
% no edges
END

% lower bound
LowerB: 4 5 2 0 0
% upper bound
UpperB: 4 8 4 7 3

NewNodeCount: 2       % number of initial nodes

SOMNeighborhoodFunctionParams:
0.05                  % alpha_winning
0.005                 % alpha_neighbors
NoPreProcessNeighborhoodFunction  % do pre-processing?

SOMTrainNetParams:    % train net parameters
10000                  % t_max
200                   % t_net
NoPreProcessTrainNet  % do pre-processing?

SOMAdaptionParams:
111                   % age_max_start
111                   % age_max_end
0.005                % beta
NoPreProcessAdaption  % do pre-processing?

NoPreProcessUpdateNode % do pre-processing?

SOMLearningRateParams:
1                     % alpha_start
0.5                   % alpha_end

SOMBreakConditionParams:
20                    % maxNodes

SOMAdaptionNetParams:
0.5                   % epsilon

END
```

Appendix C

Notation

$\mathcal{R}$	input layer (usually $\subset \mathbf{R}^n$)	
$\mathcal{C}$	output layer (usually $\subset \mathbf{R}^m$)	
N_R	dimension of the input space	
N_C	dimension of the output space	
x	input vector	$= [\xi_1, \xi_2, \ldots, \xi_{N_R}]^T \in \mathcal{R}$
w_s	weight vector	$= [\mu_{s1}, \mu_{s2}, \ldots, \mu_{sn}]^T \in \mathcal{R}$
$e_j(x)$	energy function	
s	winning neuron	
$\phi_w(x)$	mapping	$\phi_w : \mathcal{R} \mapsto \mathcal{C}, x \mapsto \phi_w(x)$
h_{si}	neighborhood function	
	defining the lateral interation between neuron s and i	
$\mathcal{E}$	set of edges	
$\mathcal{N}_c$	set of neighbors of c	$= \{i \in \mathcal{C} \mid (c,i) \in \mathcal{E}\}$
Λ	kernel size	
L	maximum edge age	
E_i	local error variable for neuron i	
$P(x)$	distribution of the input data	
α	learning rate	
q	neuron with maximal error value	$= \mathrm{argmax}_{c \in \mathcal{C}} E_c$
ϵ	update parameter for error variables	
β	update parameter for error variables	

Bibliography

[1] R. Brause (1995), Neuronale Netze: eine Einfhrung in die Neurinformatik, Stuttgart: Teubner.

[2] G. Rigoll (1994), Neuronale Netze: eine Einfhrung fr Ingenieure, Informatiker und Naturwissenschaftler, Renningen-Malmsheim: expert-Verl.

[3] Rojas (1993), Theorie der neuronalen Netze, Berlin: Springer Verlag.

[4] H. Ritter, T. Martinetz, K. Schulten (1990), Neuronale Netze, Bonn: Addison-Wesley.

[5] J. Pearl (1984), Heurisitcs: Intelligent Search Strategies for Computer Problem Solving, California: Addison-Weslay.

[6] P. Schefe (1991), Knstliche Intelligenz - berblick und Grundlagen, Mannheim: BI-Wiss.-Verl.

[7] C. von der Malsburg (1973), Self-organization of orientation sensitive cells in the striate cortex, Kybernetik,14: 85-100.

[8] S. Amari (1977), Neural Theory of Association and Concept-Foramtion, Biological Cybernetics Vol. 26: 175-185.

[9] S. Amari (1978), Mathematical Theory on formation of category detecting nerve cells, Biological Cybernetics Vol. 27: 77-87.

[10] T. Tacheuchi, S. Amari (1979), Formation of topographic maps and columnar microstructure in nerve fields, Biological Cybernetics Vol. 3: 63-72.

[11] Anderberg, Michael R. (1973), Cluster analysis for applications, New York Acad. Press.

[12] Kohonen T. (1982), Self-Organized Formation of Topologically Correct Feature Maps, Biological Cybernetics Vol. 44: 135-140.

[13] Kohonen T. (1988), Learning Vector Quantization, Abstracts of First annual INNS meeting: Pergamont Press p.303.

[14] Kohonen T. (1988b), The “Neural” Phonetic Typewriter of Helsinki University of Technology, IEEE Computer.

[15] B. Angniol, G. de la Croix Vabois, J.-Y. le Texier (1988), Self-Organizing Feature Maps and the Traveling Salesman Problem, New York: Pergamon Press, Neural Networks Vol 1: 289-293.

[16] Kohonen T. (1995), Self-Organizing Maps, Berlin: Springer Verlag.

[17] H. Ritter, K. Schulten (1988), Convergence Properties of Kohonen’s Topology conserving Maps, Biological Cybernetics Vol. 60: 59ff.

[18] L. Xu, A. Krzyzak, E. Oja (1993), Rival Penalized Competitive Learning for Clustering Analysis, RBF Net and Curve Detection, IEEE Transactions on Neural Networks, Vol. 4: 636-649.

[19] Preparata F. P. and Shamos M. I. (1990), Computational geometry, New York: Springer Verlag.

[20] Hujun Yin and Nigel M. Allinson (1995). On the distribution and convergence of feature space in self-organizing maps. Neural Computation, 7(6):1178-1187.

[21] Martinetz (1992), Selbstorganisierende neuronale Netzwerkmodelle zur Bewegungssteuerung, St. Augustin: Infix.

[22] T. M. Martinetz and K. J. Schulten (1991), A “neural-gas” network learns topologies. In T. Kohonen, K. Mkisara, O. Simula, and J. Kangas, editors, Artificial Neural Networks, pages 397-402. North-Holland, Amsterdam.

[23] B. Fritzke (1993), Growing cell structures–a self-organizing network for unsupervised and supervised learning. Technical Report TR-93-026, Int. Computer Science Institute, Berkeley, CA.

[24] T. M. Martinetz and K. J. Schulten (1994) Topology representing networks. New York: Pergamon Press, Neural Networks, 7(3):-522.

[25] B. Fritzke (1995), A growing neural gas network learns topologies. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, Advances in Neural Information Processing Systems 7, pages 625-632. MIT Press, Cambridge MA.

[26] B. Fritzke (1995), Growing grid–a self-organizing network with constant neighbourhood range and adaptation strength. Neural Processing Letters, 2(5):9-13.

[27] B. Fritzke (1996), Growing Self-organizing Networks - Why? ESANN’96, pages 61-72. D-Facto Publishers, Brussels.

[28] H.-U. Bauer, Th. Villmann (1995), Growing a Hypercubial Output Space in a Self-Organizing Feature Map Technical Report TR-95-030, Int. Computer Science Institute, Berkeley, CA.

[29] Ultsch, A. and Siemon, H. P. (1990) Kohonen's self organizing feature maps for exploratory data analysis. In Proceedings of ICNN'90, International Neural Network Conference, pages 305-308, Kluwer, Dordrecht.

[30] Kraaijveld, M. A., Mao, J., and Jain, A. K. (1992) A non-linear projection method based on Kohonen's topology preserving maps. In Proceedings of 11ICPR, 11th International Conference on Pattern Recognition, pages 41-45, IEEE Computer Society Press, Los Alamitos, CA.