UNIVERSITY OF STUTTGART

INSTITUTE FOR PARALLEL AND DISTRIBUTED HIGH PERFORMANCE SYSTEMS (IPVR)

Applied Computer Science – Image Understanding

SNNS

Stuttgart Neural Network Simulator

User Manual, Version 4.0

UNIVERSITY OF STUTTGART

INSTITUTE FOR PARALLEL AND DISTRIBUTED HIGH PERFORMANCE SYSTEMS (IPVR)

Applied Computer Science – Image Understanding

Andreas Zell
Günter Mamier, Michael Vogt
Niels Mache, Ralf Hübner, Sven Döring
Kai-Uwe Herrmann, Tobias Soyez, Michael Schmalzl
Tilman Sommer, Artemis Hatzigeorgiou, Dietmar Posselt
Tobias Schreiner, Bernward Kett, Gianfranco Clemente

external contributions by
Martin Reczko, Martin Riedmiller
Mark Seemann, Marcus Ritt
Jamie DeCoster, Jochen Biedermann
Joachim Danz, Christian Wehrfritz
Randolf Werner, Michael Berthold

SNNS

Stuttgart Neural Network Simulator

User Manual, Version 4.0

Report No. 6/95

Contents

Chapter 1

Introduction to SNNS

SNNS (Stuttgart Neural Network Simulator) is a simulator for neural networks developed at the Institute for Parallel and Distributed High Performance Systems (Institut für Parallele und Verteilte Höchstleistungsrechner, IPVR) at the University of Stuttgart since 1989. The goal of the project is to create an efficient and flexible simulation environment for research on and application of neural nets.

The SNNS simulator consists of four main components that are depicted in figure 1.1: Simulator kernel, graphical user interface, batch simullator version snnsbat, and network compiler snns2c. There was also a fifth part, Nessus, that was used to construct networks for SNNS. Nessus, however, has become obsolete since the introduction of powerful interactive network creation tools within the graphical user interface and is no longer supported. The simulator kernel operates on the internal network data structures of the neural nets and performs all operations on them. The graphical user interface XGUI[1], built on top of the kernel, gives a graphical representation of the neural networks and controls the kernel during the simulation run. In addition, the user interface can be used to directly create, manipulate and visualize neural nets in various ways. Complex networks can be created quickly and easily. Nevertheless, XGUI should also be well suited for unexperienced users, who want to learn about connectionist models with the help of the simulator. An online help system, partly context-sensitive, is integrated, which can offer assistance with problems.

An important design concept was to enable the user to select only those aspects of the visual representation of the net in which he is interested. This includes depicting several aspects and parts of the network with multiple windows as well as suppressing unwanted information.

SNNS is implemented completely in ANSI-C. The simulator kernel has already been tested on numerous machines and operating systems (see also table 1.1). XGUI is based upon X11 Release 5 from MIT and the Athena Toolkit, and was tested under various window managers, like `twm, tvtwm, olwm, ctwm, fvwm`. It also works under X11R6.

[1] X Graphical User Interface

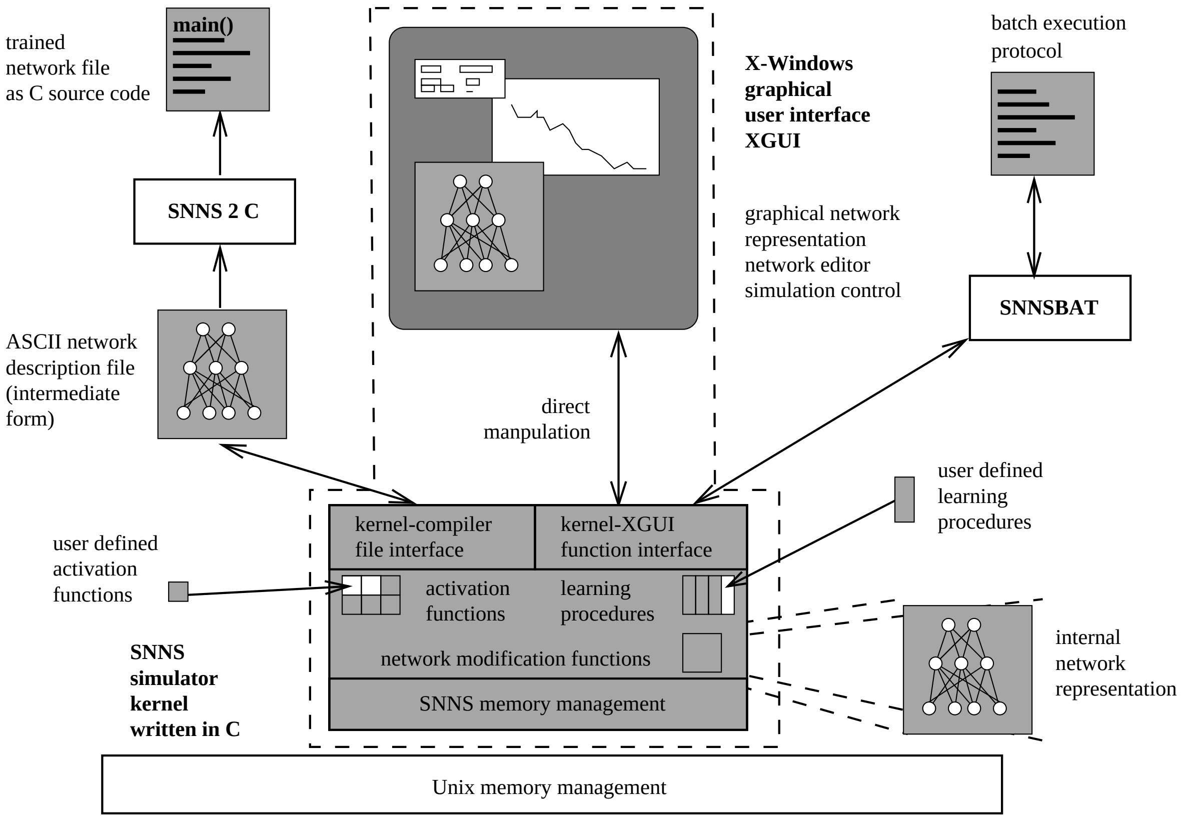

Figure 1.1: SNNS components: simulator kernel, graphical user interface xgui, snnsbat, and network compiler snns2c

machine type	**operating system**
SUN SparcSt. ELC,IPC	SunOS 4.1.2, 4.1.3
SUN SparcSt. 2	SunOS 4.1.2
SUN SparcSt. 5, 10, 20	SunOS 4.1.3, 5.2
DECstation 3100, 5000	Ultrix V4.2
DEC Alpha AXP 3000	OSF1 V2.1
IBM-PC 80486, Pentium	Linux
IBM RS 6000/320, 320H, 530H	AIX V3.1, AIX V3.2
HP 9000/720, 730	HP-UX 8.07
SGI Indigo 2	IRIX 4.0.5, 5.3

Table 1.1: Machines and operating systems on which SNNS has been tested (as of May 1995)

This document is structured as follows:

This chapter 1 gives a brief introduction and overview of SNNS.

Chapter 2 gives the details about how to obtain SNNS and under what conditions. It includes licensing, copying and exclusion of warranty. It then discusses how to install SNNS and gives acknowledgments of its numerous authors.

Chapter 3 introduces the components of neural nets and the terminology used in the description of the simulator. Therefore, this chapter may also be of interest to people already familiar with neural nets.

Chapter 4 describes how to operate the two-dimensional graphical user interface. After a short overview of all commands a more detailed description of these commands with an example dialog is given.

Chapter 5 describes the form and usage of the patterns of SNNS

Chapter 6 describes the integrated graphical editor of the 2D user interface. These editor commands allow the interactive construction of networks with arbitrary topologies.

Chapter 7 is about a tool to facilitate the generation of large, regular networks from the graphical user interface.

Chapter 8 describes the network analyzing facilities, built into SNNS.

Chapter 9 describes the connectionist models that are already implemented in SNNS, with a strong emphasis on the less familiar network models.

Chapter 10 describes the four pruning functions which are available in SNNS.

Chapter 11 introduces a new visualization component for three-dimensional visualization of the topology and the activity of neural networks with wireframe or solid models.

Chapter 12 describes how to use the full power of multiple computers in a cluster of workstations for training networks with SNNS.

Chapter 13 introduces the batch capabilities of SNNS. They can be accessed via an additional interface to the kernel, that allows for easy background execution.

Chapter 14 gives a brief overlook over the tools that come with SNNS, without being an internal part of it.

Chapter 15 describes in detail the interface between the kernel and the graphical user interface. This function interface is important, since the kernel can be included in user written C programs.

Chapter 16 details the activation functions and output function that are already built in.

In appendix A the format of the file interface to the kernel is described, in which the nets are read in and written out by the kernel. Files in this format may also be generated by any other program, or even an editor.

The grammars for both network and pattern files are also given here.

In appendix B and C examples for network and batch configuration files are given.

Chapter 2

Licensing, Copying and Acknowledgments

SNNS is ©(Copyright) 1990-95 SNNS Group, Institute for Parallel and Distributed High-Performance Systems (IPVR), University of Stuttgart, Breitwiesenstrasse 20-22, 70565 Stuttgart, Fed. Rep. of Germany.

SNNS is distributed by the University of Stuttgart as 'Free Software' in a licensing agreement similar in some aspects to the GNU General Public License. There are a number of important differences, however, regarding modifications and distribution of SNNS to third parties. Note also that SNNS is not part of the GNU software nor is any of its authors connected with the Free Software Foundation. We only share some common beliefs about software distribution. Note further that SNNS is NOT PUBLIC DOMAIN.

The SNNS License is designed to make sure that you have the freedom to give away verbatim copies of SNNS, that you receive source code or can get it if you want it and that you can change the software for your personal use; and that you know you can do these things.

We protect your and our rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy and distribute the unmodified software or modify it for your own purpose.

In contrast to the GNU license we do not allow modified copies of our software to be distributed. You may, however, distribute your modifications as separate files (e.g. patch files) along with our unmodified SNNS software. We encourage users to send changes and improvements which would benefit many other users to us so that all users may receive these improvements in a later version. The restriction not to distribute modified copies is also useful to prevent bug reports from someone else's modifications.

Also, for our protection, we want to make certain that everyone understands that there is NO WARRANTY OF ANY KIND for the SNNS software.

2.1 SNNS License

1. This License Agreement applies to the SNNS program and all accompanying programs and files that are distributed with a notice placed by the copyright holder saying it may be distributed under the terms of the SNNS License. "SNNS", below, refers to any such program or work, and a "work based on SNNS" means either SNNS or any work containing SNNS or a portion of it, either verbatim or with modifications. Each licensee is addressed as "you".

2. You may copy and distribute verbatim copies of SNNS's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of SNNS a copy of this license along with SNNS.

3. You may modify your copy or copies of SNNS or any portion of it only for your own use. You may not distribute modified copies of SNNS. You may, however, distribute your modifications as separate files (e.g. patch files) along with the unmodified SNNS software. We also encourage users to send changes and improvements which would benefit many other users to us so that all users may receive these improvements in a later version. The restriction not to distribute modified copies is also useful to prevent bug reports from someone else's modifications.

4. If you distribute copies of SNNS you may not charge anything except the cost for the media and a fair estimate of the costs of computer time or network time directly attributable to the copying.

5. You may not copy, modify, sub-license, distribute or transfer SNNS except as expressly provided under this License. Any attempt otherwise to copy, modify, sub-license, distribute or transfer SNNS is void, and will automatically terminate your rights to use SNNS under this License. However, parties who have received copies, or rights to use copies, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

6. By copying, distributing or modifying SNNS (or any work based on SNNS) you indicate your acceptance of this license to do so, and all its terms and conditions.

7. Each time you redistribute SNNS (or any work based on SNNS), the recipient automatically receives a license from the original licensor to copy, distribute or modify SNNS subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein.

8. Incorporation of SNNS or parts of it in commercial programs requires a special agreement between the copyright holder and the Licensee in writing and usually involves the payment of license fees. If you want to incorporate SNNS or parts of it in commercial programs write to the author about further details.

9. Because SNNS is licensed free of charge, there is no warranty for SNNS, to the extent permitted by applicable law. The copyright holders and/or other parties provide SNNS "as is" without warranty of any kind, either expressed or implied,

including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of SNNS is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.

10. In no event will any copyright holder, or any other party who may redistribute SNNS as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use SNNS (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of SNNS to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

2.2 How to obtain SNNS

The SNNS simulator can be obtained via anonymous ftp from host

```
ftp.informatik.uni-stuttgart.de  (129.69.211.2)
```

in the subdirectory

```
/pub/SNNS
```

as file

```
SNNSv4.0.tar.Z
```

or in several parts as files

```
SNNSv4.0.tar.Z.aa, SNNSv4.0.tar.Z.ab, ...
```

These split files are each less than 1 MB and can be joined with the Unix ‘cat’ command into one file `SNNSv4.0.tar.Z`. Be sure to set the ftp mode to `binary` before transmission of the files. Also watch out for possible higher version numbers, patches or Readme files in the above directory `/pub/SNNS`. After successful transmission of the file move it to the directory where you want to install SNNS, uncompress the file with the Unix command

```
uncompress SNNSv4.0.tar.Z
```

and then use the command

```
tar -xvf SNNSv4.0.tar
```

This will extract SNNS in the current directory. The SNNS distribution includes full source code, installation procedures for supported machine architectures and some simple examples of trained networks. The full English documentation as LaTeX source code with PostScript images included and a PostScript version of the documentation is also available in the SNNS directory.

2.3 Installation

Note that SNNS has not been tested extensively in different computer environments and is a research tool with frequent substantial changes. It should be obvious that we don't guarantee anything. We are also not staffed to answer problems with SNNS or to fix bugs quickly.

Starting with release 3.0, SNNS needs an ANSI-C compiler to run. The Kernighan & Ritchie standard is no longer supported.

To build SNNS in the directory in which you have moved it you first have to generate the correct makefiles for your machine architecture and window system used. To do this, simply call the shell script

```
configure
```

This prompts you for information about the machine architecture and the window system and builds all necessary makefiles with this information. It uses templates found in the directory configuration for this task. Under certain conditions the shell script may not be able to traverse logical links. If you are sure about the location of your X11 libraries and include files, but configure can't locate them, respond by giving "unknown" as path. You can later set the correct path in the Makefiles with any text editor. At the end of this script, you will be told about the next step to build the simulator.

The next step to build the simulator is usually to build the kernel and the graphical user interface with the command

```
build sim
```

or with the two commands

```
build kernel
build xgui
```

This script descends into the appropriate subdirectories and calls the makefiles in these subdirectories to compile all necessary source files and link the object files into one executable file. The executable is located in

```
<SNNS-dir.>/xgui/bin/<architecture>/xgui
```

where <SNNS-dir.> is the current SNNS directory and <architecture> is the machine architecture, e.g. dec_alpha,, hp_ux, sun_os4x, or others.

We usually build a symbolic link named snns to point to this executable if we often work on the same machine architecture.

```
ln -s  xgui/bin/<architecture>/xgui  snns
```

This link may also be placed in the user home directory (with the proper path prefix to SNNS) or in a directory of binaries in the local users' search path.

The simulator is then called simply with

```
snns
```

If the tools accompanying the simulator are also to be built, use the command

```
build all
```

which calls the three commands

```
build kernel
build xgui
build tools
```

Important Note:

Since version 3.2, the SNNS pattern files are read with the help of a parser generator. However, the standard compiler-compiler for unix, yacc, is not powerful enough for this task, and the GNU utility program bison has to be used. For those sites that do not have bison installed, we have a precompiled file included in the distribution. These circumstances have two important consequences:

- The precompiled file kr_pat_parse.c was generated on a DECStation. We have found that some compilers on other machines refuse to process this file for various reasons. This is true for all combinations of bison-machine_A / gcc-machine_B pairs, but the file generated by the DEC bison seems to be the most widely accepted.
- If you have a bison on your target machine, the first installation of SNNS will overwrite our precompiled file. This is fine as long as only this machine is used. If later on SNNS is to be compiled for another machine though, this new file will be used if the second machine does not have bison. In this case, the compatibility problems mentioned above might block a successful compilation.
- If you do not have bison installed and your compiler does not like the precompiled file either, you have to install bison. Bison is a GNU product and is not distributed by us. Please refer to the various GNU ftp sites to obtain this program.

For further details about calling the simulator and working with the graphical user interface see chapter 4.

2.4 Acknowledgments

SNNS is a joint effort of a number of people, computer science students, research assistants as well as faculty members at the University of Stuttgart, Institute for Parallel and Distributed High Performance Systems (IPVR), Stuttgart, Germany.

The project to develop an efficient and portable neural network simulator which later became SNNS was lead since 1989 by Dr. Andreas Zell, who designed the predecessor to the SNNS simulator and the SNNS simulator itself and acted as advisor for more than

two dozen independent research and Master's thesis projects that made up the SNNS simulator and some of its applications. Over time the SNNS source grew to a total size of now 5.6MB in 190.000 lines of code. All the research has been done under the supervision of Prof. Dr. Andreas Reuter and Prof. Dr. Paul Levi. We are all grateful for their continuing support and for providing us with the necessary computer and network equipment.

The following persons were directly involved in the SNNS project. They are listed in the order in which they joined the SNNS team.

Andreas Zell	Design of the SNNS simulator, SNNS project team leader [ZMS90], [ZMSK91b] [ZMSK91c], [ZMSK91a]
Niels Mache	SNNS simulator kernel (really the heart of SNNS) [Mac90], parallel SNNS kernel on MasPar MP-1216.
Tilman Sommer	original version of the graphical user interface XGUI with integrated network editor [Som89], PostScript printing.
Ralf Hübner	SNNS simulator 3D graphical user interface [Hüb92], user interface development (version 2.0 to 3.0).
Thomas Korb	SNNS network compiler and network description language Nessus [Kor89]
Michael Vogt	Radial Basis Functions [Vog92]. Together with Günter Mamier implementation of Time Delay Networks. Definition of the new pattern format.
Günter Mamier	SNNS visualization and analyzing tools [Mam92]. Implementation of the batch execution capability. Together with Michael Vogt implementation of the new pattern handling. Compilation and continuous update of the user manual. Maintenance of the ftp server. Installation of external contributions.
Michael Schmalzl	SNNS network creation tool Bignet, implementation of Cascade Correlation, and printed character recognition with SNNS [Sch91a]
Kai-Uwe Herrmann	ART models ART1, ART2, ARTMAP and modification of the BigNet tool [Her92].
Artemis Hatzigeorgiou	Video documentation about the SNNS project, learning procedure Backpercolation 1.[1]
Dietmar Posselt	ANSI-C translation of SNNS.
Sven Döring	ANSI-C translation of SNNS and source code maintenance. Implementation of distributed kernel for workstation clusters.

[1] Backpercolation 1 was developed by JURIK RESEARCH & CONSULTING, PO 2379, Aptos, CA 95001 USA. Any and all SALES of products (commercial, industrial, or otherwise) that utilize the Backpercolation 1 process or its derivatives require a license from JURIK RESEARCH & CONSULTING. Write for details.

Tobias Soyez	Jordan and Elman networks, implementation of the network analyzer [Soy93].
Tobias Schreiner	Network pruning algorithms [Sch94]
Bernward Kett	Redesign of C-code generator snns2c
Gianfranco Clemente	Help with the user manual
Henri Bauknecht	Manager of the SNNS mailing list.

We are proud of the fact that SNNS is experiencing growing support from people outside our university. There are many people who helped us by pointing out bugs or offering bug fixes, both to us and other users. Unfortunately they are to numerous to list here, so we restrict ourselves to those who have made a major contribution to the source code.

Martin Riedmiller, University of Karlsruhe
Implementation of RPROP in SNNS

Martin Reczko, German Cancer Research Center (DKFZ)
Implementation of Backpropagation Through Time (BPTT), BatchBackpropagation Through Time (BBPTT), and Quickprop Through Time (QPTT).

Mark Seemann and Marcus Ritt, University of Tübingen
Implementation of self organizing maps.

Jamie DeCoster, Purdue University
Implementation of auto-associative memory functions.

Jochen Biedermann, University of Göttingen
Help with the implementation of pruning Algorithms and non-contributing units

Christian Wehrfritz, University of Erlangen
Original implementation of the projection tool, implementation of the statistics computation and learning algorithm Pruned Cascade Correlation.

Randolf Werner, University of Koblenz
Support for NeXT systems

Joachim Danz, University of Darmstadt
Implementation of cross validation, simulated annealing and Monte Carlo learning algorithms.

Michael Berthold, University of Karlsruhe
Implementation of enhanced RBF algorithms

If you would like to contact the SNNS team via e-mail please write to Andreas Zell at 'zell@informatik.uni-stuttgart.de'.

The SNNS simulator is a successor to an earlier neural network simulator called NetSim [ZKSB89], [KZ89] by A. Zell, T. Sommer, T. Korb and A. Bayer, which was itself influenced by the popular Rochester Connectionist Simulator RCS [GLML89].

In September 1991 the Stuttgart Neural Network Simulator SNNS was awarded the "Deutscher Hochschul-Software-Preis 1991" (German Federal Research Software Prize) by the German Federal Minister for Science and Education, Prof. Dr. Ortleb.

2.5 New Features of Release 4.0

Users already familiar with SNNS and its usage may be interested in the differences between the versions 3.3 and 4.0. New users of SNNS may skip this section and proceed with the next chapter.

New Features of Release 4.0:

1. A new distributed version of the kernel. SNNS can now be spread out in a workstation cluster for faster learning.
2. Improved version of the C-code generator snns2c
3. Validation sets now can check on the performance of the network during training.
4. New error function in tool analyze that can handle single-output-unit-networks.
5. New statistic tool that can predict the generalization capabilities of the network.
6. Reorganization of the manager panel with new selection mechanism for the major SNNS windows.
7. The remote panel was changed significantly and renamed 'control panel'.
8. Selection possibility in the graph panel between plot of SSE, MSE, and SSE/output.
9. New learning algorithm RBF_DDA
10. New learning algorithm simulated annealing
11. New learning algorithm Monte Carlo.
12. New learning algorithm Pruned-Cascade-Correlation
13. Several new initialization algorithms for Kohonen and Counterpropagation networks, since the two old ones had serious flaws.
14. Key codes to bring up all the main panels of SNNS (e.g Alt-c for the control panel).
15. Support for NeXT systems.
16. More information printed to the shell during training and with the INFO button.
17. The SOURCE and TARGET part of the info panel have been swapped for a more intuitive setup of the information.
18. Extensive debugging (as usual).

Chapter 3

Neural Network Terminology

Connectionism is a current focus of research in a number of disciplines, among them artificial intelligence (or more general computer science), physics, psychology, linguistics, biology and medicine. Connectionism represents a special kind of information processing: Connectionist systems consist of many primitive cells (*units*) which are working in parallel and are connected via directed links (*links, connections*). The main processing principle of these cells is the distribution of activation patterns across the links similar to the basic mechanism of the human brain, where information processing is based on the transfer of activation from one group of neurons to others through synapses. This kind of processing is also known as *parallel distributed processing (PDP)*.

The high performance of the human brain in highly complex cognitive tasks like visual and auditory pattern recognition was always a great motivation for modeling the brain. For this historic motivation connectionist models are also called *neural nets*. However, most current neural network architectures do not try to closely imitate their biological model but rather can be regarded simply as a class of parallel algorithms.

In these models, knowledge is usually distributed throughout the net and is stored in the structure of the topology and the weights of the links. The networks are organized by (automated) training methods, which greatly simplify the development of specific applications. Classical logic in ordinary AI systems is replaced by vague conclusions and associative recall (exact match vs. best match). This is a big advantage in all situations where no clear set of logical rules can be given. The inherent fault tolerance of connectionist models is another advantage. Furthermore, neural nets can be made tolerant against noise in the input: with increased noise, the quality of the output usually degrades only slowly (*graceful performance degradation*).

3.1 Building Blocks of Neural Nets

The following paragraph describes a generic model for those neural nets that can be generated by the SNNS simulator. The basic principles and the terminology used in dealing with the graphical interface are also briefly introduced. A more general and more detailed introduction to connectionism can, e.g., be found in [RM86]. For readers fluent in German, the most comprehensive and up to date book on neural network learning algorithms, simulation systems and neural hardware is probably [Zel94]

A network consists of *units*[1] and directed, weighted *links* (connections) between them. In analogy to activation passing in biological neurons, each unit receives a net input that is computed from the weighted outputs of prior units with connections leading to this unit. Picture 3.1 shows a small network.

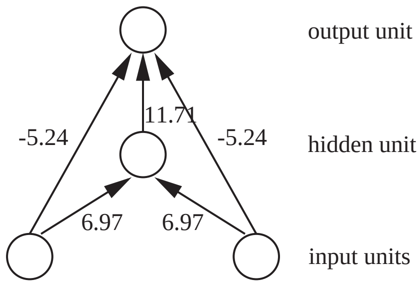

Figure 3.1: A small network with three layers of units

The actual information processing within the units is modeled in the SNNS simulator with the *activation function* and the *output function*. The activation function first computes the net input of the unit from the weighted output values of prior units. It then computes the new activation from this net input (and possibly its previous activation). The output function takes this result to generate the output of the unit.[2] These functions can be arbitrary C functions linked to the simulator kernel and may be different for each unit.

Our simulator uses a discrete clock. Time is not modeled explicitly (i.e. there is no propagation delay or explicit modeling of activation functions varying over time). Rather, the net executes in *update steps*, where $a(t+1)$ is the activation of a unit one step after $a(t)$.

The SNNS simulator, just like the Rochester Connectionist Simulator (RCS, [God87]), offers the use of *sites* as additional network element. Sites are a simple model of the dendrites of a neuron which allow a grouping and different treatment of the input signals of a cell. Each site can have a different site function. This selective treatment of incoming information allows more powerful connectionist models. Figure 3.2 shows one unit with sites and one without.

In the following all the various network elements are described in detail.

3.1.1 Units

Depending on their function in the net, one can distinguish three types of units: The units whose activations are the problem input for the net are called *input units*; the units

[1] In the following the more common name "units" is used instead of "cells".

[2] The term *transfer function* often denotes the combination of activation and output function. To make matters worse, sometimes the term activation function is also used to comprise activation and output function.

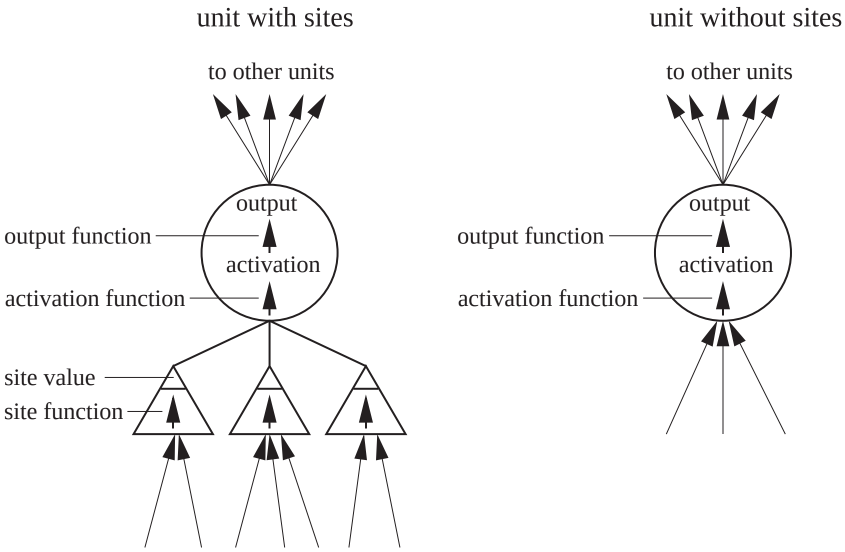

Figure 3.2: One unit with sites and one without

whose output represent the output of the net *output units*. The remaining units are called *hidden units*, because they are not visible from the outside (see e.g. figure 3.1).

In most neural network models the type correlates with the topological position of the unit in the net: If a unit does not have input connections but only output connections, then it is an input unit. If it lacks output connections but has input units, it is an output unit. If it has both types of connections it is a hidden unit.

It can, however, be the case that the output of a topologically internal unit is regarded as part of the output of the network. The IO-type of a unit used in the SNNS simulator has to be understood in this manner. That is, units can receive input or generate output even if they are not at the fringe of the network.

Below, all *attributes* of a unit are listed:

- `no`: For proper identification, every unit has a number[3] attached to it. This number defines the order in which the units are stored in the simulator kernel.
- `name`: The name can be selected arbitrarily by the user. It must not, however, contain blanks or special characters, and has to start with a letter. It is useful to select a short name that describes the task of the unit, since the name can be displayed with the network.
- `io-type` or `io`: The IO-type defines the function of the unit within the net. The following alternatives are possible
  - *input*: input unit
  - *output*: output unit
  - *dual*: both input and output unit

[3]This number can change after saving but remains unambiguous. See also chapter 4.2.2.1

  - *hidden*: internal, i.e. hidden unit
  - *special*: this type can be used in any way, depending upon the application. In the standard version of the SNNS simulator, the weights to such units are not adapted in the learning algorithm (see paragraph 3.3).
  - *special input, special hidden, special output*: sometimes it is necessary to to know where in the network a special unit is located. These three types enable the correlation of the units to the various layers of the network.

- `activation`: The activation value.
- `initial activation` or `i_act`: This variable contains the initial activation value, present after the initial loading of the net. This initial configuration can be reproduced by resetting (*reset*) the net, e.g. to get a defined starting state of the net.
- `output`: the output value.
- `bias`: In contrast to other network simulators where the bias (threshold) of a unit is simulated by a link weight from a special 'on'-unit, SNNS represents it as a unit parameter. In the standard version of SNNS the bias determines where the activation function has its steepest ascent. (see e.g. the activation function `Act_logistic`). Learning procedures like backpropagation change the bias of a unit like a weight during training.
- `activation function` or `actFunc`: A new activation is computed from the output of preceding units, usually multiplied by the weights connecting these predecessor units with the current unit, the old activation of the unit and its bias. When sites are being used, the network input is computed from the site values. The general formula is:

$$a_j(t+1) = f_{act}(net_j(t), a_j(t), \theta_j)$$

  where:

$a_j(t)$	activation of unit j in step t
$net_j(t)$	net input in unit j in step t
θ_j	threshold (bias) of unit j

  The SNNS default activation function *Act_logistic*, for example, computes the network input simply by summing over all weighted activations and then squashing the result with the logistic function $f_{act}(x) = 1/(1+e^{-x})$. The new activation at time $(t+1)$ lies in the range $[0,1]$[4]. The variable θ_j is the threshold of unit j.

  The net input $net_j(t)$ is computed with

$$\begin{aligned} net_j(t) &= \sum_i w_{ij} o_i(t) \quad \text{if unit } j \text{ has no sites} \\ net_j(t) &= \sum_k s_{jk}(t) \quad \text{if the unit } j \text{ has sites, with site values} \end{aligned}$$

[4] Mathematically correct would be $]0,1[$, but the values 0 and 1 are reached due to arithmetic inaccuracy.

$$s_{jk}(t) = \sum_i w_{ij} o_i(t)$$

This yields the well-known logistic activation function

$$a_j(t+1) = \frac{1}{1 + e^{-(\sum_i w_{ij} o_i(t) - \theta_j)}}$$

where:

$a_j(t)$	activation of unit j in step t
$net_j(t)$	net input in unit j in step t
$o_i(t)$	output of unit i in step t
$s_{jk}(t)$	site value of site k on unit j in step t
j	index for some unit in the net
i	index of a predecessor of the unit j
k	index of a site of unit j
w_{ij}	weight of the link from unit i to unit j
θ_j	threshold (bias) of unit j

Activation functions in SNNS are relatively simple C functions which are linked to the simulator kernel. The user may easily write his own activation functions in C and compile and link them to the simulator kernel. How this can be done is described later.

- `output function` or `outFunc`: The output function computes the output of every unit from the current activation of this unit. The output function is in most cases the identity function (SNNS: `Out_identity`). This is the default in SNNS. The output function makes it possible to process the activation before an output occurs.

$$o_j(t) = f_{out}(a_j(t))$$

where:

$a_j(t)$	Activation of unit j in step t
$o_j(t)$	Output of unit j in step t
j	Index for all units of the net

Another predefined SNNS-standard function, *Out_Clip01* clips the output to the range of [0..1] and is defined as follows:

$$o_j(t) = \begin{cases} 0 & \text{if } a_j(t) < 0 \\ 1 & \text{if } a_j(t) > 1 \\ a_j(t) & \text{otherwise} \end{cases}$$

Output functions are even simpler C functions than activation functions and can be user-defined in a similar way.

- `f-type`: The user can assign so called f-types (functionality types, prototypes) to a unit. The unusual name is for historical reasons. One may think of an f-type as a pointer to some prototype unit where a number of parameters has already been defined:
  - activation function and output function
  - whether sites are present and, if so, which ones

  These types can be defined independently and are used for grouping units into sets of units with the same functionality. All changes in the definition of the f-type consequently affect also all units of that type. Therefore a variety of changes becomes possible with minimum effort.
- `position`: Every unit has a specific position (coordinates in space) assigned to it. These positions consist of 3 integer coordinates in a 3D grid. For editing and 2D visualization only the first two (x and y) coordinates are needed, for 3D visualization of the networks the z coordinate is necessary.
- `subnet no`: Every unit is assigned to a subnet. With the use of this variable, structured nets can be displayed more clearly than would otherwise be possible in a 2D presentation.
- `layers`: Units can be visualized in 2D in up to 8 layers[5]. Layers can be displayed selectively. This technique is similar to a presentation with several transparencies, where each transparency contains one aspect or part of the picture, and some or all transparencies can be selected to be stacked on top of each other in a random order. Only those units which are in layers (transparencies) that are 'on' are displayed. This way portions of the network can be selected to be displayed alone. It is also possible to assign one unit to multiple layers. Thereby it is feasible to assign any combination of units to a layer that represents an aspect of the network.
- `frozen`: This attribute flag specifies that activation and output are frozen. This means that these values don't change during the simulation.

All 'important' unit parameters like activation, initial activation, output etc. and all function results are computed as floats with nine decimals accuracy.

3.1.2 Connections (Links)

The direction of a connection shows the direction of the transfer of activation. The unit from which the connection starts is called the *source unit*, or *source* for short, while the other is called the *target unit*, or *target*. Connections where source and target are identical (recursive connections) are possible. Multiple connections between one unit and the same input port of another unit are redundant, and therefore prohibited. This is checked by SNNS.

Each connection has a *weight* (or strength) assigned to it. The effect of the output of one unit on the successor unit is defined by this value: if it is negative, then the connection

[5] Changing it to 16 layers can be done very easily in the source code of the interface.

is inhibitory, i.e. decreasing the activity of the target unit; if it is positive, it has an excitatory, i.e. activity enhancing, effect.

The most frequently used network architecture is built hierarchically bottom-up. The input into a unit comes only from the units of preceding layers. Because of the unidirectional flow of information within the net they are also called feed-forward nets (as example see the neural net classifier introduced in chapter 3.5). In many models a full connectivity between all units of adjoining levels is assumed.

Weights are represented as floats with nine decimal digits of precision.

3.1.3 Sites

A unit with sites doesn't have a direct input any more. All incoming links lead to different sites, where the arriving weighted output signals of preceding units are processed with different user-definable site functions (see picture 3.2). The result of the site function is represented by the site value. The activation function then takes this value of each site as network input.

The SNNS simulator does not allow multiple connections from a unit to the same input port of a target unit. Connections to different sites of the same target units are allowed. Similarly, multiple connections from one unit to different input sites of itself are allowed as well.

3.2 Update Modes

To compute the new activation values of the units, the SNNS simulator running on a sequential workstation processor has to visit all of them in some sequential order. This order is defined by the *Update Mode*. Five update modes for general use are implemented in SNNS. The first is a synchronous mode, all other are asynchronous, i.e. in these modes units see the new outputs of their predecessors if these have fired before them.

1. *synchronous*: The units change their activation all together after each step. To do this, the kernel first computes the new activations of all units from their activation functions in some arbitrary order. After all units have their new activation value assigned, the new output of the units is computed. The outside spectator gets the impression that all units have fired simultaneously (in sync).

2. *random permutation*: The units compute their new activation and output function sequentially. The order is defined randomly, but each unit is selected exactly once in every step.

3. *random*: The order is defined by a random number generator. Thus it is not guaranteed that all units are visited exactly once in one update step, i.e. some units may be updated several times, some not at all.

4. *serial*: The order is defined by ascending internal unit number. If units are created with ascending unit numbers from input to output units, this is the fastest mode.

Note that the use of serial mode is not advisable if the units of a network are not in ascending order.

5. *topological*: The kernel sorts the units by their topology. This order corresponds to the natural propagation of activity from input to output. In pure feed-forward nets the input activation reaches the output especially fast with this mode, because many units already have their final output which doesn't change later.

Additionally, there are 12 more update modes for special network topologies implemented in SNNS.

1. *CPN*: For learning with counterpropagation.
2. *Time Delay*: This mode takes into account the special connections of time delay networks. Connections have to be updated in the order in which they become valid in the course of time.
3. *ART1_Stable*, *ART2_Stable* and *ARTMAP_Stable*: Three update modes for the three adaptive resonance theory network models. They propagate a pattern through the network until a stable state has been reached.
4. *ART1_Synchronous*, *ART2_Synchronous* and *ARTMAP_Synchronous*: Three other update modes for the three adaptive resonance theory network models. They perform just one propagation step with each call.
5. *CC* and *RCC*: Special update modes for the cascade correlation and recurrent cascade correlation meta algorithms.
6. *BPTT*: For recurrent networks, trained with 'backpropagation through time'.
7. *RM_Synchronous*: Special update mode for auto-associative memory networks.

Note, that all update modes only apply to the *forward propagation phase*, the backward phase in learning procedures like backpropagation is not affected at all.

3.3 Learning in Neural Nets

An important focus of neural network research is the question of how to adjust the weights of the links to get the desired system behavior. This modification is very often based on the Hebbian rule, which states that a link between two units is strengthened if both units are active at the same time. The Hebbian rule in its general form is:

$$\Delta w_{ij} = g(a_j(t), t_j)h(o_i(t), w_{ij})$$

where:

w_{ij}	weight of the link from unit i to unit j
$a_j(t)$	activation of unit j in step t
t_j	teaching input, in general the desired output of unit j
$o_i(t)$	output of unit i at time t

$g(\ldots)$ function, depending on the activation of the unit and the teaching input
$h(\ldots)$ function, depending on the output of the preceding element and the current weight of the link

Training a feed-forward neural network with supervised learning consists of the following procedure:

An input pattern is presented to the network. The input is then propagated forward in the net until activation reaches the output layer. This constitutes the so called *forward propagation phase*.

The output of the output layer is then compared with the teaching input. The error, i.e. the difference (delta) δ_j between the output o_j and the teaching input t_j of a target output unit j is then used together with the output o_i of the source unit i to compute the necessary changes of the link w_{ij}. To compute the deltas of inner units for which no teaching input is available, (units of hidden layers) the deltas of the following layer, which are already computed, are used in a formula given below. In this way the errors (deltas) are propagated backward, so this phase is called *backward propagation*.

In *online learning*, the weight changes Δw_{ij} are applied to the network after each training pattern, i.e. after each forward and backward pass. In *offline learning* or *batch learning* the weight changes are cumulated for all patterns in the training file and the sum of all changes is applied after one full cycle (epoch) through the training pattern file.

The most famous learning algorithm which works in the manner described is currently backpropagation. In the backpropagation learning algorithm online training is usually significantly faster than batch training, especially in the case of large training sets with many similar training examples.

The backpropagation weight update rule, also called *generalized delta-rule* reads as follows:

$$\begin{aligned}
\Delta w_{ij} &= \eta\, \delta_j\, o_i \\
\delta_j &= \begin{cases} f'_j(net_j)(t_j - o_j) & \text{if unit } j \text{ is a output-unit} \\ f'_j(net_j)\sum_k \delta_k w_{jk} & \text{if unit } j \text{ is a hidden-unit} \end{cases}
\end{aligned}$$

where:

η learning factor eta (a constant)
δ_j error (difference between the real output and the teaching input) of unit j
t_j teaching input of unit j
o_i output of the preceding unit i
i index of a predecessor to the current unit j with link w_{ij} from i to j
j index of the current unit
k index of a successor to the current unit j with link w_{jk} from j to k

There are several backpropagation algorithms supplied with SNNS: one "vanilla backpropagation" called `Std_Backpropagation`, one with momentum term and flat spot elimination called `BackpropMomentum` and a batch version called `BackpropBatch`. They can be chosen from the **control** panel with the button `OPTIONS` and the menu selection `select learning function`.

In SNNS, one may either set the number of training cycles in advance or train the network until it has reached a predefined error on the training set.

3.4 Generalization of Neural Networks

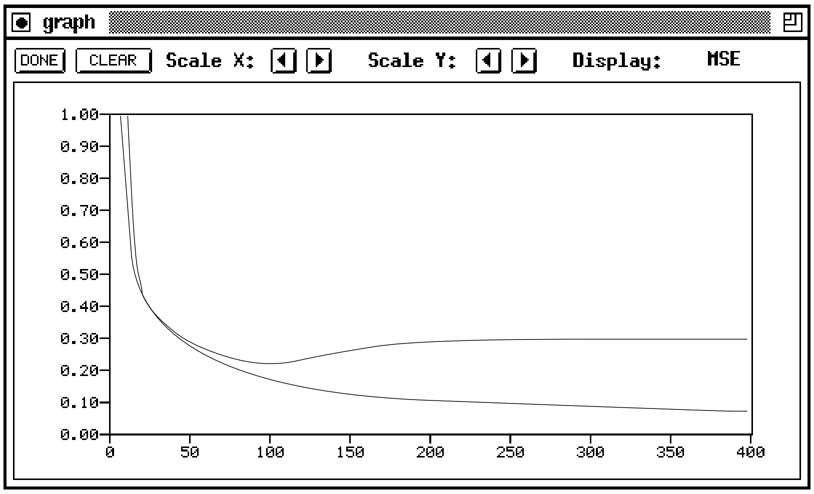

Figure 3.3: Error development of a training and a validation set

One of the major advantages of neural nets is their ability to generalize. This means that a trained net could classify data from the same class as the learning data that it has never seen before. In real world applications developers normally have only a small part of all possible patterns for the generation of a neural net. To reach the best generalization, the dataset should be split into three parts:

- The **training set** is used to train a neural net. The error of this dataset is minimized during training.
- The **validation set** is used to determine the performance of a neural network on patterns that are not trained during learning.
- A **test set** for finally checking the over all performance of a neural net.

Figure 3.3 shows a typical error development of a training set (lower curve) and a validation set (upper curve).

The learning should be stopped in the minimum of the validation set error. At this point the net generalizes best. When learning is not stopped, overtraining occurs and the performance of the net on the whole data decreases, despite the fact that the error on the training data still gets smaller. After finishing the learning phase, the net should be finally checked with the third data set, the test set.

SNNS performs one validation cycle every n training cycles. Just like training, validation is controlled from the control panel.

3.5 An Example of a simple Network

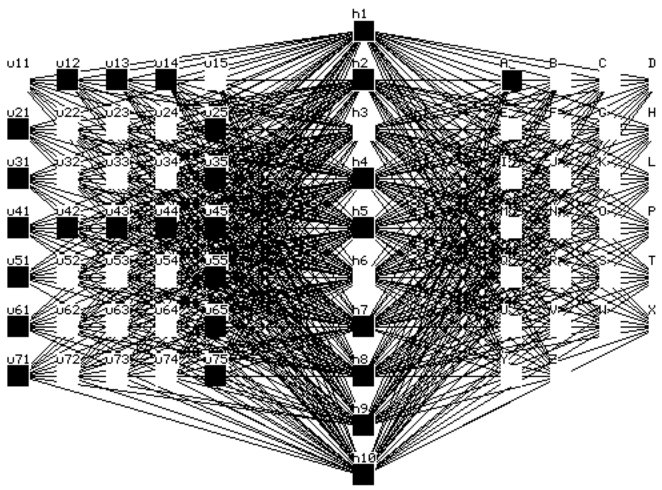

Figure 3.4: Example network of the letter classifier

This paragraph describes a simple example network, a neural network classifier for capital letters in a 5x7 matrix, which is ready for use with the SNNS simulator. Note that this is a toy example which is not suitable for real character recognition.

- Network-Files: `letters0.net` (untrained), `letters.net` (trained)
- Pattern-File: `letters.pat`

The network in figure 3.4 is a feed-forward net with three layers of units (two layers of weights) which can recognize capital letters. The input is a 5x7 matrix, where one unit is assigned to each pixel of the matrix. An activation of +1.0 corresponds to "pixel set", while an activation value of 0.0 corresponds to "pixel not set". The output of the network consists of exactly one unit for each capital letter of the alphabet.

The following activation function and output function are used by default:

- Activation function: `Act_logistic`
- Output function: `Out_identity`

The net has one input layer (5x7 units), one hidden layer (10 units) and one output layer (26 units named 'A' ...'Z'). The total of $(35 \cdot 10 + 10 \cdot 26) = 610$ connections form the distributed memory of the classifier.

On presentation of a pattern that resembles the uppercase letter "A", the net produces as output a rating of which letters are probable.

Chapter 4

Using the Graphical User Interface

This chapter describes how to use XGUI, the X-Window based graphical user interface to SNNS. XGUI is the usual way to interact with SNNS on Unix workstations. It tells how to call SNNS and explains the multiple windows and their buttons and menus. Together with the chapters 5 and 6 it is probably the most important chapter in this manual.

4.1 XGUI Files

The graphical user interface consists of the following files:

`snns` or `xgui`	SNNS simulator program (XGUI and simulator kernel linked together into one executable program)
`default.cfg`	default configuration (see chapter 4.2.2)
`help.hdoc`	help text used by XGUI

The file `snns` in the home directory of SNNS is only a symbolic link to the file

`xgui/bin/<architecture>/xgui`

where `<architecture>` is one of the currently supported machine architectures, like `dec_alpha`, `dec_mips`, `hp_ux`, `ibm_aix`, `sgi_irix`, `pc_linux`, `sun_os4x`, or `sun_solaris`.

The file `Readme_xgui` contains changes performed after printing of this document. The user is urged to read it, prior to using XGUI. The file `help.hdoc` is explained in chapter 4.2.10.

XGUI looks for the files `default.cfg` and `help.hdoc` first in the current directory. If not found there, it looks in the directory specified by the environment variable `XGUILOADPATH`. By the command

`setenv XGUILOADPATH` *Path*

this variable can be set to the path where `default.cfg` and `help.hdoc` are located. This is best done by an entry to the files `.login` or `.cshrc`. Advanced users may change the help file or the default configuration for their own purposes. This should be done, however, only on a copy of the files in a private directory.

SNNS uses the following extensions for its files:

`.net`	network files (units and link weights)
`.pat`	pattern files
`.cfg`	configuration settings files
`.txt`	text files (log files)
`.res`	result files (unit activations)

A simulator run is started by the command

```
snns [<netfile>.net] [<pattern>.pat] [<config>.cfg] [options] Return
```

where valid options are

-font `<name>` : font for the simulator
-dfont `<name>` : font for the displays
-mono : black & white on color screens
-help : help screen to explain the options

in the home directory of SNNS or by directly calling

```
<SNNS-directory>/xgui/bin/<architecture>/xgui
```

from any directory. Note that the shell variable XGUILOADPATH must be set properly before, or SNNS will complain about missing files `default.cfg` and `help.hdoc`.

The executable `xgui` may also be called with X-Window parameters as arguments.

Setting the display font can be advisable, if the font selected by the SNNS automatic font detection looks ugly. The following example starts the display with the 7x13bold font

```
snns -font 7x13bold Return
```

The fonts which are available can be detected with the program xfontsel (not part of this distribution). The current version of SNNS can not handle fonts wider than 8 pixels.

4.2 Windows of XGUI

The graphical user interface has the following windows, which can be positioned and handled independently (*toplevel shells*):

- **Manager** panel with buttons to open other windows, a message line, and a line with status information at the bottom .
- **File** browser for loading and saving networks and pattern files.
- **Control** panel for simulator operations.
- **Info** panel for setting and getting information about unit and link attributes.
- several **Displays**, to display the network graphically in two dimensions.

- **3D View** panel to control the three dimensional network visualization component.
- **Graph** display, to explain the network error during teaching graphically.
- **Bignet** panel to facilitate the creation of big regular networks.
- **Pruning** panel for control of the pruning algorithm.
- **Cascade** panel for control of the learning phase of cascade correlation learning.
- **Kohonen** panel, an extension to the control panel for Kohonen networks.
- **Weight Display**, to show the weight matrix as a WV- or Hinton diagram.
- **Projection** panel to clarify the influence of two units onto a third one.
- **Analyzer** for supervising recurrent (and other) networks.
- **Inversion** display, to control the inversion method network analyzing tool.
- **Print** panel to generate a PostScript picture of one of the 2D displays.
- **Help windows** to display the help text.

Of these windows only the Manager panel and possibly one or more 2D displays are open from the start, the other windows are opened with the corresponding buttons in the manager panel or by giving the corresponding key code while the mouse pointer is in one of the SNNS windows.

Additionally, there are several popup windows (*transient shells*) which only become visible when called and block all other XGUI windows. Among them are various **Setup** panels for adjustments of the graphical representation. (called with the button `SETUP` in the various windows)

There are a number of other popup windows which are invoked by pressing a button in one of the main windows or choosing a menu.

Figure 4.1 shows a typical screen setup. The **Manager** panel contains buttons to call all other windows of the interface and displays the status of SNNS. It should therefore always be kept visible.

The **Info** panel displays the attributes of two units and the data of the link between them. All attributes may also be changed here. The data displayed here is important for many editor commands.

In each of the **Displays** a part of the network is displayed, while all settings can be changed using **Setup**. These windows also allow access to the **network editor** using the keyboard (see also chapter 6).

The **Control** panel controls the simulator operations during learning and recall.

In the **File** panel a log file can be specified, where all XGUI output to `stdout` is copied to. A variety of data about the network can be displayed here. Also a record is kept on the load and save of files and on the teaching.

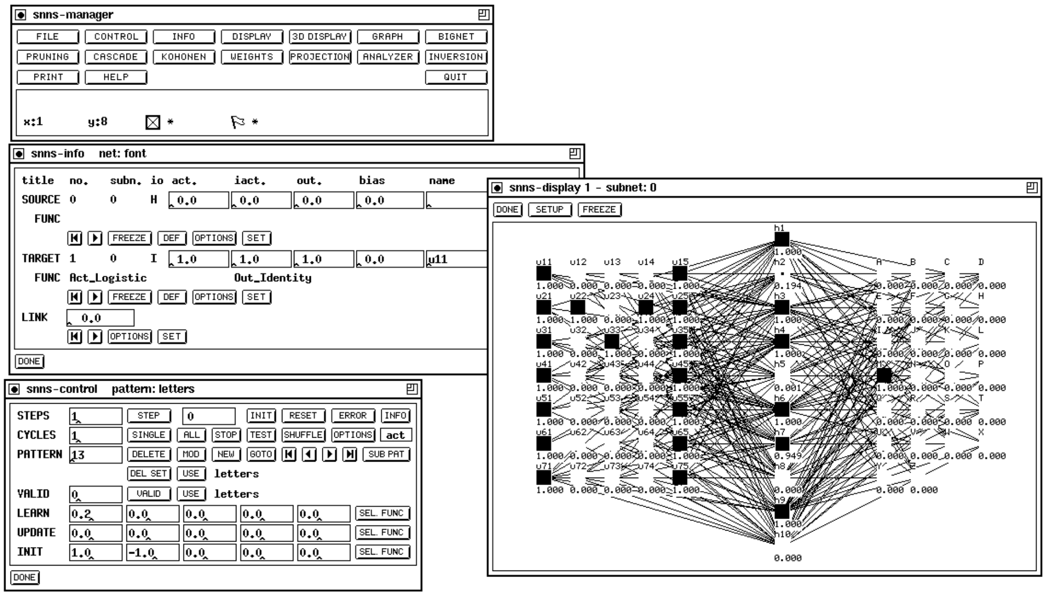

Figure 4.1: Manager panel, info panel, control panel and a display.

The complete help text from the file `help.hdoc` is available in the text section of a **help window**. Information about a word can be retrieved by marking that word in the text and then clicking `LOOK` or `MORE`. A list of keywords can be obtained by a click to `TOPICS`. This window also allows context sensitive help, when the editor is used with the keyboard.

`QUIT` is used to leave XGUI. XGUI can also be left by pressing ALT-q in any SNNS window. Pressing ALT-Q will exit SNNS without asking further questions.

4.2.1 Manager Panel

Figure 4.2 shows the manager panel. From the manager panel all other elements that have a different, independent window assigned can be called. Because this window is of such central importance, it is recommended to keep it visible all the time.

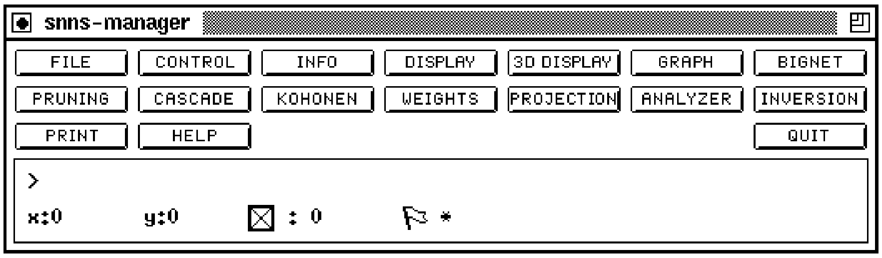

Figure 4.2: Manager panel

The user can request several displays or help windows, but only one control panel or text window. The windows called from the manager panel may also be called via key codes as follows ('Alt-' meaning the alternate key in conjunction with some other key).

FILE	Alt-f	CONTROL	Alt-c
INFO	Alt-i	DISPLAY	Alt-d
3D DISPLAY	Alt-3	GRAPH	Alt-g
BIGNET	Alt-b	PRUNING	
CASCADE		KOHONEN	Alt-k
WEIGHTS	Alt-w	PROJECTION	Alt-p
ANALYZER	Alt-a	INVERSION	
PRINT		HELP	Alt-h
QUIT	Alt-q		

SNNS Status Message:

This line features messages about a current operation or its termination. It is also the place of the command sequence display of the editor. When the command is activated, a message about the execution of the command is displayed. For a listing of the command sequences see chapter 6.

Status line

This line shows the current position of the mouse in a display, the number of selected units, and the position of flags, set by the editor.

X:0 Y:0 gives the current position of the mouse in the display in SNNS unit coordinates.

The next icon shows a small selected unit. The corresponding value is the number of currently selected units. This is important, because there might be selected units not visible in the displays. The selection of units affects only editor operations (see chapter 6 and 6.3).

The last icon shows a minature flag. If `safe` appears next to the icon, the safety flag was set by the user (see chapter 6). In this case XGUI forces the user to confirm any delete actions.

4.2.2 File Browser

The file browser handles all load and save operations of networks, patterns, configurations, and the contents of the text window. Configurations include number, location and dimension of the displays as well as their setup values and the name of the layers.

In the top line, the path (*without* trailing slash) where the files are located is entered. This can be done either manually, or by double–clicking on the list of files and directories in the box on the left. A double click to [..] deletes the last part of the path, and a double click to a subdirectory appends that directory to the path. In the input field below the path field, the name for the desired file (without extension) is entered. Again, this can be done either manually, or by double–clicking on the list of files in the box on the left. Whether a pattern file, network file, or other file is loaded/saved depends on the settings

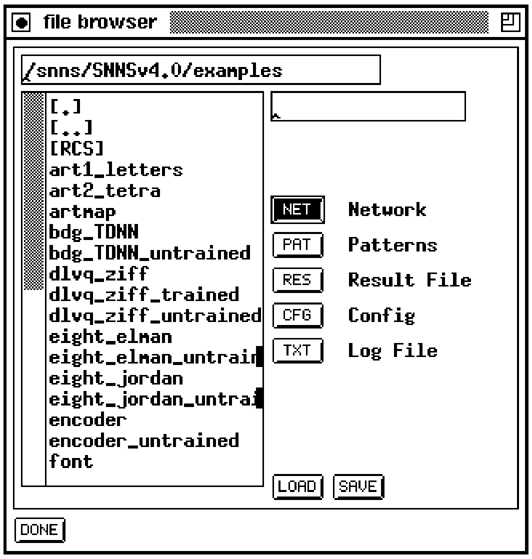

Figure 4.3: File Panel.

of the corresponding buttons below. With the setting of picture 4.3 a network file would be selected. A file name beginning with a slash (/) is taken to be an absolute path.

Note: The extension `.net` for nets, `.pat` for patterns, `.cfg` for configurations, and `.txt` for texts is added automatically and *must not* be specified. After the name is specified the desired operation is selected by clicking either `LOAD` or `SAVE`. In the case of an error the confirmer appears with an appropriate message. These errors might be:

Load:	The file does not exist or has the wrong type
Save:	A file with that name already exists

Depending upon the error and the response to the confirmer, the action is aborted or executed anyway.

NOTE: The directories must be executable in order to be processed properly by the program!

4.2.2.1 Loading and Saving Networks

If the user wants to load a network which is to replace the net in main memory, the confirmer appears with the remark that the current network would be erased upon loading. If the question `'Load?'` is answered with `YES`, the new network is loaded. The file name of the network loaded last appears in the window title of the manager panel.

Note 1: Upon saving the net the kernel compacts its internal data structures if the units are not numbered consecutively. This happens if units are deleted during the creation of the network. All earlier listings with unit numbers then become invalid. The user is therefore advised to save and reload the network after creation, before continuing the work.

Note 2: The assignment of patterns to input or output units may be changed after a network save, if an input or output unit is deleted and is inserted again. This is caused because the activation values in the pattern file are assigned to units in ascending order of the unit number. However, this order is no longer the same because the new input or output units may have been assigned higher unit numbers than the existing input or output units. So some components of the patterns may be assigned incorrectly.

4.2.2.2 Loading and Saving Patterns

Patterns are combinations of activations of input or output units. Pattern files, like nets, are handled by the SNNS kernel. Upon loading the patterns, it is not checked whether the patterns fit to the network. If the number of activation values does not fit to the number of input, resp. output units, a sub-pattern shifting scheme has to be defined later on in the sub-pattern panel. See chapter 5 for details. The filename of the patterns loaded last is displayed in the control panel.

Note: The activation values are read and assigned to the input and output units sequentially in ascending order of the unit numbers (see above).

4.2.2.3 Loading and Saving Configurations

A configuration contains the location and size of all displays with all setup parameters and the names of the various layers. This information can be loaded and saved separately, since it is independent from the networks. Thereby it is possible to define one configuration for several networks, as well as several configurations for the same net. When XGUI is started, the file `default.cfg` is loaded automatically.

4.2.2.4 Saving a Result file

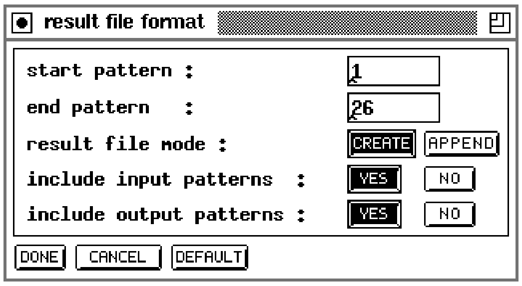

Figure 4.4: Result File Popup

A result file contains the activations of all output units. These activations are obtained by performing one pass of forward propagation. After pressing the SAVE button a popup window lets the user select which patterns are to be tested and which patterns are to be saved in addition to the test output. Picture 4.4 shows that popup window. Since the result file has no meaning for the loaded network a load operation is not useful and therefore not supported.

4.2.2.5 Defining the Log File

Messages that document the simulation run can be stored in the **log file**. The protocol contains file operations, definitions of values set by clicking the `SET` button in the info panel, as well as a teaching protocol (cycles, parameters, errors). In addition, the user can output data about the network to the log file with the help of the info panel. If no log file is loaded, output takes place only on `stdout`. If no file name is specified when clicking `LOAD`, a possibly open log file is closed and further output is restricted to `stdout`.

4.2.3 Control Panel

With this window the simulator is operated. Figure 4.5 shows this window. Table 4.1 lists all the input options with types and value ranges. The meaning of the 5 learning parameters depends upon the learning function selected with the menu `select learning function` invoked by the button `OPTIONS` of the control panel.

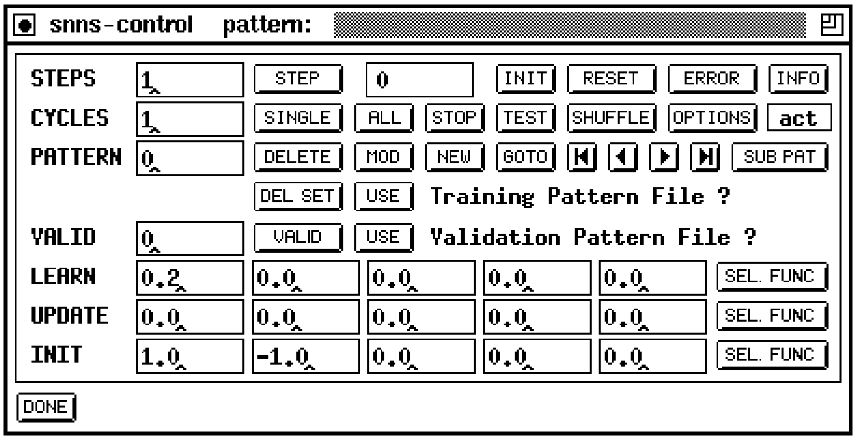

Figure 4.5: Control Panel

Name	Type	Value Range
`STEPS` (update-Steps)	Text	$0 \le n$
`COUNT` (counter for steps)	Label	$0 \le n$
`CYCLES`	Text	$0 \le n$
`PATTERN` (number of current pattern)	Label	$0 \le n$
`VALID`	Text	$0 \le n$
`LEARN` (5 parameters: η, α, d, ...)	Text	float
`UPDATE` (5 parameters)	Text	float
`INIT` (5 parameters)	Text	float

Table 4.1: Input fields of the control panel

There are the following text fields, buttons and menu buttons:

1. `STEPS`: This text field specifies the number of update steps of the network. With `Topological_Order` selected as update mode (chosen with the menu `select update function` from the button `OPTIONS` in the control panel) one step is sufficient to propagate information from input to output. With other update modes or with recursive networks, several steps might be needed.

2. `STEP`: After clicking this button, the simulator kernel executes the number of steps specified in the text field `STEPS`. If `STEPS` is zero, the units are only redrawn. The update mode selected with the button `MODE` is used (see chapter 3.2). The first update step in the mode `topological` takes longer than the following, because the net is sorted topologically first. Then all units are redrawn.

3. `COUNT`: The text field next to the `STEP` button displays the steps, executed so far.

4. `INIT`: Initializes the network with values according to the function and parameters given in the last line of the panel.

5. `RESET`: The counter is reset and the units are assigned their initial activation.

6. `ERROR`: By pressing the error button in the control panel, SNNS will print out several statistics. The formulas were contributed by Warren Sarle from the SAS institute. Note that these criteria are for *linear* models; they can sometimes be applied directly to nonlinear models if the sample size is large. A recommended reference for linear model selection criteria is [JGHL80].

   Notation:

n	=	Number of observations (sample size)
p	=	Number of parameters, to be estimated (i.e. weights)
SSE	=	The sum of squared errors
TSS	=	The total sum of squares corrected for the mean for the dependent variable

   Criteria for adequacy of the estimated model in the sample
   Pearson's R^2, the proportion of variance, is explained or accounted by the model:
   $$R^2 := 1 - \frac{SSE}{TSS}$$

   Criteria for adequacy of the true model in the population
   The mean square error [JGHL80] is defined as: $MSE := \frac{SSE}{n-p}$, the root mean square error as: $RMSE := \sqrt{MSE}$.

   The R^2_{adj}, the R^2 [JGHL80] adjusted for degrees of freedom, is defined as:
   $$R^2_{adj} := 1 - \frac{n-1}{n-p} \cdot (1 - R^2)$$

   Criteria for adequacy of the estimated model in the population
   Anemiya's prediction criterion [JGHL80] is similar to the R^2_{adj}:
   $$PC := 1 - \frac{n+p}{n-p} \cdot (1 - R^2)$$

   The estimated mean square error of prediction (J_p) assuming that the values of the regressors are fixed and that the model is correct is:
   $$J_p := (n + p) \cdot MSE/n$$

   The conservative mean square error in prediction [Weh94] is:
   $$CMSEP := \frac{SSE}{n-2p}$$

The generalized cross validation (GCV) is given by Wahba [GHW79] as:

$$GCV := \frac{SSE \cdot n}{(n-p)^2}$$

The estimated mean square error of prediction assuming that both independent and dependent variables are multivariate normal is defined as:

$$GMSEP := \frac{MSE(n+1)(n-2)}{n(n-p-1)}$$

Shibata's criterion $SHIBATA := \frac{SSE(n+2p)}{n}$ can be found in [Shi68].

Finally, there is Akaikes information criterion [JGHL80]:

$$AIC := n \cdot \ln \frac{SSE}{n} + 2p$$

and the Schwarz's Bayesian criterion [JGHL80]:

$$SBC := n \cdot \ln \frac{SSE}{n} + n \cdot \ln p.$$

Obviously, most of these selection criteria do only make sense, if $n >> p$.

7. `INFO`: Information about the net is written to the text window.

8. `CYCLES`: This text field specifies the number of learning cycles. It is mainly used in conjunction with the next two buttons. A cycle (also called an epoch sometimes) is a unit of training where all patterns of a pattern file are presented to the network once.

9. `SINGLE`: The net is trained with a single pattern for the number of training cycles defined in the field `CYCLES`. The text window reports the error of the network every `CYCLES`/10 cycles, i.e. independent of the number of training cycles only 10 numbers are generated. (This prevents flooding the user with network performance data and slowing down the training by file I/O).

   The error reported in the text window is the sum of the quadratic differences between the teaching input and the real output over all output units. The average error per output unit is given behind `ave`.

10. `ALL`: The net is trained with all patterns for the number of training cycles specified in the field `CYCLES`. This is the usual way to train networks from the graphical user interface. Note, that if cycles has a value of, say, 100, the button `ALL` causes SNNS to train all patterns once (one cycle = one epoch) and repeat this 100 times (NOT training each pattern 100 times in a row and then applying the next pattern).

    The error reported in the text window is the sum of the quadratic difference between the teaching input and the real output over all output units summed over the number of patterns presented. The average error per output unit is given behind `ave`.

11. `STOP`: Stops the teaching cycle. After completion of the current step or teaching cycle, the simulation is halted immediately.

12. `TEST`: With this button, the user can test the behavior of the net with all patterns loaded. The activation values of input and output units are copied into the net. (For output units see also button `SHOW`). Then the number of update steps specified in `STEPS` are executed.

13. `SHUFFLE`: It is important for optimal learning that the various patterns are presented in different order in the different cycles. A random sequence of patterns is

created automatically, if `SHUFFLE` is switched on.

14. `OPTIONS`: Offers the following menu:

`jog weights`	change all link weights randomly
`edit f-types`	edit/create f-types
`edit sites`	edit/create sites

If `jog weights` is selected, a popup window appears to specify the value range (`low limit .. high limit`) of the random noise to be added to all links in the network.

15. `act`: With this button, the user specifies the changes to the activation values of the output units when a pattern is applied with `TEST`. The following table gives the three possible alternatives:

`None`	The output units remain unchanged.
`Out`	The output values are computed and set, activations remain unchanged.
`Act`	The activation values are set.

The label of this button always displays the item selected from the menu.

16. `PATTERN`: This text field displays the current pattern number.

17. `DELETE`: The pattern whose number is displayed in the text field `PATTERN` is deleted from the pattern file.

18. `MOD`: The pattern whose number is displayed in the text field `PATTERN` is modified in place.

    The current activation of the input units and the current output values of output units of the network loaded make up the input and output pattern. These values might have been set with the network editor and the Info panel before.

19. `NEW`: A new pattern is defined that is added behind existing patterns. Input and output values are defined as above. This button is disabled whenever the current pattern set has variable dimensions.

20. `GOTO`: The simulator advances to the pattern whose number is displayed in the text field `PATTERN`.

21. Arrow buttons ⏮, ◀, ▶, and ⏭: With these buttons, the user can navigate through all patterns loaded, as well as jump directly to the first and last pattern. Unlike with the button `TEST` no update steps are performed here.

22. `SUB PAT`: Opens the panel for sub-pattern handling. The button is inactive when the current pattern set has no variable dimensions. The sub-pattern panel is described in section 5.3.

23. `DEL SET`: Opens the menu of loaded pattern sets. The pattern set of the selected entry is removed from main memory. The corresponding pattern file remains untouched. When the current pattern set is deleted, the last in the list becomes current. When the last remaining pattern set is deleted, the current pattern set becomes undefined and the menu shows the entry `No Files`.

24. `USE`: Also opens the menu of loaded pattern sets. The pattern set of the selected entry becomes the current set. All training, testing, and propagation actions refer always to the current pattern set. The name of the corresponding pattern file is displayed next to the button in the *Current Pattern Set* field.

25. *Current Pattern Set*: This field displays the name of the pattern set currently used for training. When no current pattern set is defined the entry "`Training Pattern File ?`" is displayed.

26. `VALID`: Gives the intervals in which the training process is to be interrupted by the computation of the error on the validation pattern set. A value of 0 inhibits validation. The validation error is printed on the shell window and plotted in the graph display.

27. `USE`: Opens the menu of loaded pattern sets. The pattern set of the selected entry becomes the current validation set. The name of the corresponding pattern file is displayed next to the button in the *Validation Pattern Set* field.

28. *Validation Pattern Set*: This field displays the name of the pattern set currently used for validation. When no current pattern set is defined the entry "`Validation Pattern File ?`" is displayed.

29. `LEARN`: Five fields to specify the parameters of the learning function. The number required and their resp. meaning depend upon the learning function used. A description of the learning functions that are already built in into SNNS is given in section 4.3.

30. `SEL. FUNC`: in the LEARN row invokes a menu to select a learning function (learning procedure). The following learning functions are currently implemented:

`ART1`	ART1 learning algorithm
`ART2`	ART2 learning algorithm
`ARTMAP`	ARTMAP learning algorithm
	(all ART models by Carpenter & Grossberg)
`BBPTT`	Batch-Backpropagation for recurrent networks
`BPTT`	Backpropagation for recurrent networks
`Backpercolation`	Backpercolation 1 (Mark Jurik)
`BackpropBatch`	Backpropagation for batch training
`BackpropMomentum`	Backpropagation with momentum term
`BackpropWeightDecay`	Backpropagation with Weight Decay
`CC`	Cascade correlation meta algorithm
`Counterpropagation`	Counterpropagation (Robert Hecht-Nielsen)
`Dynamic_LVQ`	LVQ algorithm with dynamic unit allocation
`Hebbian`	Hebbian learning rule
`JE_BP`	Backpropagation for Jordan-Elman networks
`JE_BP_Momentum`	BackpropMomentum for Jordan-Elman networks
`JE_Quickprop`	Quickprop for Jordan-Elman networks
`JE_Rprop`	Rprop for Jordan-Elman networks

`Kohonen`	Kohonen Self Organizing Maps
`Monte-Carlo`	Monte-Carlo learning
`PruningFeedForward`	Pruning algorithms
`QPTT`	Quickprop for recurrent networks
`Quickprop`	Quickprop (Scott Fahlman)
`RCC`	Cascade Correlation for recurrent networks
`RM_delta`	Rumelhart-McClelland's delta rule
`RadialBasisLearning`	Radial Basis Functions
`RBF-DDA`	modified Radial Basis Functions
`Rprop`	Resilient Propagation learning
`SimAnn_SS_error`	Simulated Annealing with SSE computation
`SimAnn_WTA_error`	Simulated Annealing with WTA computation
`SimAnn_WWTA_error`	Simulated Annealing with WWTA computation
`Std_Backpropagation`	"vanilla" Backpropagation
`TimeDelayBackprop`	Backpropagation for TDNNs (Alex Waibel)

31. `UPDATE`: Five fields to specify the parameters of the update function. The number required and their resp. meaning depend upon the update function used.

32. `SEL. FUNC`: in the UPDATE row invokes a menu to select an update function. A list of the update functions that are already built in into SNNS and their descriptions is given in section 4.4.

33. `INIT`: Five fields to specify the parameters of the init function. The number required and their resp. meaning depend upon the init function used.

34. `SEL. FUNC`: in the INIT row invokes a menu to select an initialization function. See section 4.5 for a list of the init functions available as well as their description.

4.2.4 Info Panel

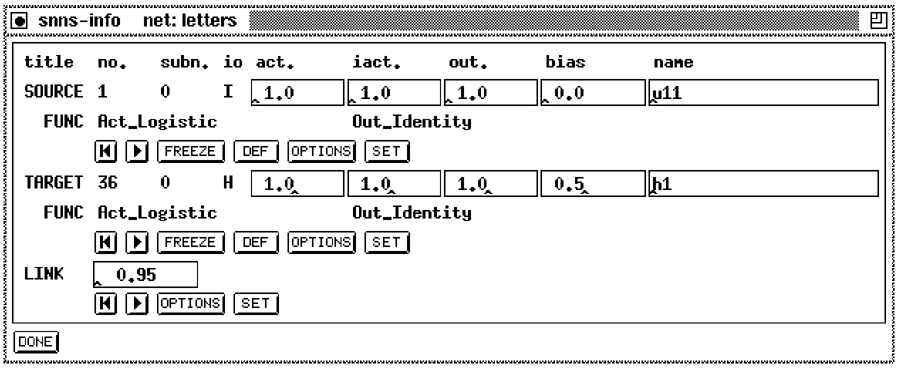

Figure 4.6: Info panel

The info panel displays all data of two units and the link between them. The unit at the beginning of the link is called `SOURCE`, the other `TARGET`. One may run sequentially through all connections or sites of the `TARGET` unit with the arrow buttons and look at the corresponding source units and vice versa.

This panel is also very important for editing, since some operations refer to the displayed `TARGET` unit or (`SOURCE`$\rightarrow$`TARGET`) link. A default unit can also be created here, whose values (activation, bias, IO-type, subnet number, layer numbers, activation function, and output function) are copied into all selected units of the net.

The source unit of a link can also be specified in a 2D display by pressing the middle mouse button, the target unit by releasing it. To select a link between two units the user presses the middle mouse button on the source unit in a 2D display, moves the mouse to the target unit while holding down the mouse button and releases it at the target unit. Now the selected units and their link are displayed in the info panel. If no link exists between two units selected in a 2D display, the `TARGET` is displayed with its first link, thereby changing `SOURCE`.

In table 4.2 the various fields are listed. The fields in the second line of the SOURCE or TARGET unit display the name of the activation function, name of the output function, name of the f-type (if available). The fields in the line `LINK` have the following meaning: weight, site value, site function, name of the site. Most often only a link weight is available. In this case no information about sites is displayed.

Name	Meaning	Type	set by	value range
no.	unit no.	Label		$1..2^{31}$
subn.	subnet no.	Label		$-32736..32735$
io	IO-type	Label	`OPTIONS`	I(nput), O(utput), H(idden), D(ual), S(pecial)
act.	activation	Text	input	float;
iact.	initial act.	Text	input	float;
out.	output	Text	input	float;
bias	bias value	Text	input	float
name	unit name	Text	input	string, starting with letter
activation function		Label	`OPTIONS`	as available
output function		Label	`OPTIONS`	as available
link	weight	Text	input	float
site value		Label		float
site function		Label		as available
site name		Label		as available at `TARGET`

Table 4.2: Table of the unit, link and site fields in the Info panel

Unit number, unit subnet number, site value, and site function cannot be modified. To change attributes of type text, the cursor has to be exactly in the corresponding field.

There are the following buttons for the units (from left to right):

1. Arrow button : The button below TARGET selects the first target unit (of the given source unit); the button below SOURCE selects the first source unit (of the given target unit);
2. Arrow button : The button below TARGET selects the next target unit (of the given source unit); the button below SOURCE selects the next source unit (of the

given target unit);

3. `FREEZE`: Unit is frozen, if this button is inverted. Changes become active only after `SET` is clicked.
4. `DEF`: The default unit is assigned the displayed values of `TARGET` and `SOURCE` (only activation, bias, IO-type, subnet number, layer numbers, activation function and output function).
5. `OPTIONS`: Calls the following menu:

`change io-type`	change the IO-type
`change f-type`	change f-type
`display activation function`	graph of the activation function
`change activation function`	change activation function
	note: f-type gets lost!
`display output function`	graph of the output function
`change output function`	change output function
	note: f-type gets lost!
`assign layers`	assign unit to layers
`list all sources`	list all predecessors
`list all targets`	list all successors

6. `SET`: Only after clicking this button the attributes of the corresponding unit are set to the specified value. The unit is also redrawn. Therefore the values can be changed without immediate effect on the unit.

There exist the following buttons for links (from left to right):

1. : Select first link of the `TARGET` unit.
2. : Select next link of the `TARGET` unit.
3. `OPTIONS`: Calls the following menu:

`list current site of TARGET`	list of all links of current site.
`list all sites of TARGET`	list all sites of the `TARGET`
`list all links from SOURCE`	list all links starting at `SOURCE`
`delete site`	delete displayed site
	note: f-type gets lost!
`add site`	add new site to `TARGET`
	note: f-type gets lost!

4. `SET`: Only after clicking this button the link weight is set.

4.2.4.1 Unit Function Displays

The characteristic functions of the units can be displayed in a graphic representation. For this purpose separate displays have been created, that can be called by selecting the options `display activation function` or `display output function` in the menu under the `options` button of the target and source unit in the info panel.

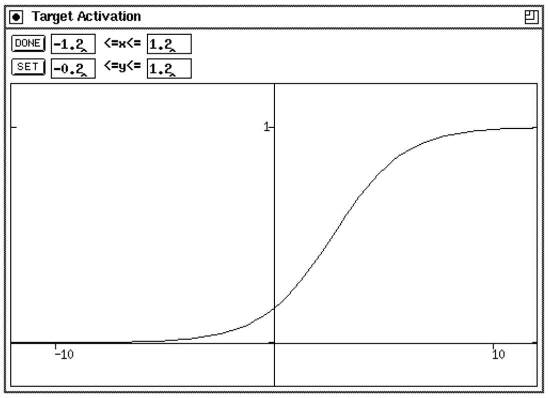

Figure 4.7: The logistic activation function in a unit function display

Figure 4.7 shows an example of an activation function. The window header states whether it is an activation or an output function, as well as whether it is the current function of the source or target unit.

The size of the window is as flexible as the picture range of the displayed function. The picture range can be changed by using the dialog widgets at the top of the function displays. The size of the window may be changed by using the standard mechanisms of your window manager.

If a new activation or output function has been defined for the unit, the display window changes automatically to reflect the new situation. Thereby it is easy to get a quick overview of the available functions by opening the function displays and then clicking through the list of available functions (This list can be obtained by selecting `select activation function` or `select output function` in the unit menu).

4.2.5 2D Displays

A 2D display or simply display is always part of the user interface. It serves to display the network topology, the units' activations and the weights of the links. Each unit is located on a grid position, which simplifies the positioning of the units. The distance between two grid points (`grid width`) can be changed from the default 37 pixels to other values in the setup panel.

The current position, i.e. the grid position of the mouse, is also numerically displayed at the bottom of the manager panel. The x-axis is the horizontal line and valid coordinates lie in the range $-32736 \ldots +32735$ (short integer).

The current version displays units as boxes, where the size of the box is proportional to the value of the displayed attribute. Possible attributes are activation, initial activation, bias, and output. A black box represents a positive value, an empty box a negative value. The size of the unit varies between 16x16 and 0 pixels according to the value of $scaleFaktor$.

The parameter *scaleFactor* has a default value of 1.0, but may be set to values between 0.0 and 2.0 in the setup panel. Each unit can be displayed with two of several attributes. One above the unit and one below the unit. The attributes to be displayed can be selected in the setup panel.

Links are shown as solid lines, with optional numerical display of the weight in the center of the line and/or arrow head pointing to the target unit. These features are optional, because they heavily affect the drawing speed of the display window.

A display can also be frozen with the button `FREEZE` (button gets inverted). It is afterwards neither updated anymore[1], nor does it accept further editor commands.

An iconified display is not updated and therefore consumes (almost) no CPU time. If a window is closed, its dimensions and setup parameters are saved in a stack (LIFO). This means that a newly requested display gets the values of the window assigned that was last closed.

For better orientation, the window title contains the subnet number which was specified for this display in the setup panel.

4.2.5.1 Setup Panel of a 2D Display

Changes to the kind of display of the network can be performed in the **Setup panel**. All settings become valid only after the button `DONE` is clicked. The whole display window is then redrawn.

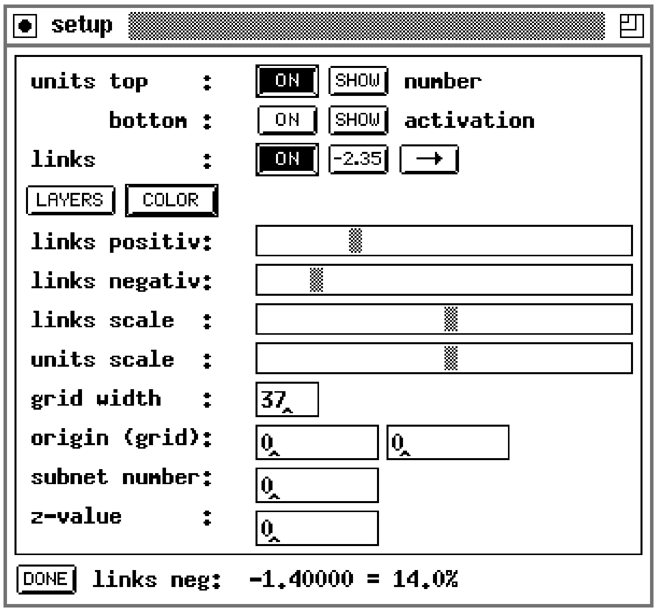

Figure 4.8: Setup Panel of a 2D-display.

[1]If a frozen display has to be redrawn, e.g. because an overlapping window was moved, it gets updated. If the network has changed since the freeze, its contents will also have changed!

1. Buttons to control the display of unit information: The first two lines of the Setup panel (`units top` and `units bottom`) contain two buttons each to set the unit parameter that can be displayed at the top resp. the bottom of the unit.

   The button `ON` toggles the display of information which can be selected with the button `SHOW`. The unit name, unit number, or the z-value (3D coordinate) can be displayed above the unit, the activation, initial activation, bias, or output of the unit below the unit. The numerical attribute selected with the button `SHOW` at the bottom of the unit (activation, initial activation, output, or bias) also determines the size of the unit in the graphical representation.

   It is usually not advisable to switch off `top` (number or name), because this information is needed for reference to the info panel. An unnamed unit is always displayed with its number.

2. Buttons to control the display of link information: The third line consists of three buttons to select the display of link data, `ON`, $\boxed{-2.35}$, $\boxed{\rightarrow}$.

   - `ON` determines whether to draw links at all (then `ON` is inverted),
   - $\boxed{-2.35}$ displays link weights at the center of the line representing the link,
   - $\boxed{\rightarrow}$ displays arrow heads of the links pointing from source to target unit.

3. `LAYERS` invokes another popup window to select the display of up to eight different layers in the display window. Layers are being stacked like transparent sheets of paper and allow for a selective display of units and links. These layers need NOT correspond with layers of units of the network topology (as in multilayer feed-forward networks), but they may do so. Layers are very useful to display only a selected subset of the network. The display of each layer can be switched on or off independently. A unit may belong to several layers at the same time. The assignment of units to layers can be done with the menu `assign layers` invoked with the button `OPTIONS` in the main Info panel.

4. `COLOR` sets the 2D–display colors. On monochrome terminals, black on white or white on black representation of the network can be selected from a popup menu. On color displays, a color editing window is opened. This window consists of three parts:

   The palette of available colors at the top, the buttons to select the item to be colored in the lower left region, and the color preview window in the lower right region.

   A color is set by clicking first at the appropriate button (`TEXT`, `BACKGROUND`, or `SELECTION`) and then at the desired color in the color palette. The selected setting is immediately displayed in the color preview window. All colors may be set in any order and any number of times. The changes become effective in the corresponding 2D–display only after both the setup panel and the color edit panel have been dismissed with the `DONE` button.

5. Sliders for the selection of link display parameters, `links positive` and `links negative`:

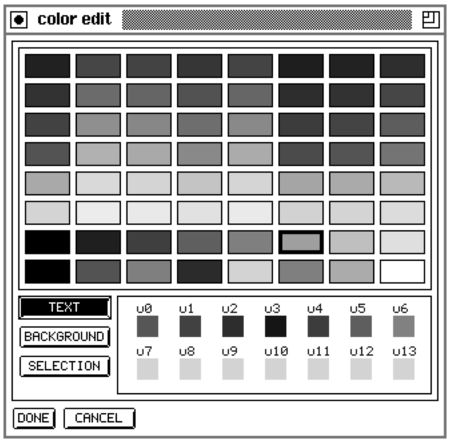

Figure 4.9: Color Setup Panel.

There are two slidebars to set thresholds for the display of links. When the bubble is moved, the current threshold is displayed in absolute and relative value at the bottom of the setup panel. Only those links with an absolute value above the threshold are displayed. The range of the absolute values is $0.0 \leq linkTrigger \leq 10.0$ (see also paragraph 4.2.5). The trigger values can be set independently for positive and negative weights. With these link thresholds the user can concentrate on the strong connections. Reducing the number of links drawn is an effective means to speed up the drawing of the displays, since line drawing takes most of the time to display a network.

Note: The links that are not drawn are only invisible. They still remain accessible, i.e. they are affected by editor operations.

6. `units scale`: This slidebar sets the parameter *scaleFactor* for the size of the growing boxes of the units. Its range is $0.0 \leq scaleFaktor \leq 2.0$. A scale factor of 0.5 draws the unit with activation 0.5 with full size. A scale factor of 2.0 draws a unit with activation 1.0 only with half size.

7. `grid width`: This value sets the width of the grid on which the units are placed. For some nets, changing the default of 37 pixels may be useful, e.g. to be able to better position the units in a geometrical pattern. Overlapping tops and bottoms occur if a grid size of less than 35 pixels is selected (26 pixels if units are displayed without numerical values). This overlap, however, does not affect computation in any way.

8. `origin (grid)`: These two fields determine the origin of the window, i.e. the grid position of the top left corner. There, the left field represents the x coordinate, the right is the y coordinate. The origin is usually (0, 0). Setting it to (20, 0) moves the display 20 units to the right and 10 units down in the grid.

9. `subnet number`: This field adjusts the subnet number to be displayed in this window. Values between -32736 and $+32735$ are possible here.

4.2.6 Graph Window

Graph is a tool to visualize the error development of a net. The program is started by clicking the `graph` button in the manager panel or by typing Alt-g in any SNNS window. Figure 4.10 shows the window of the graph tool.

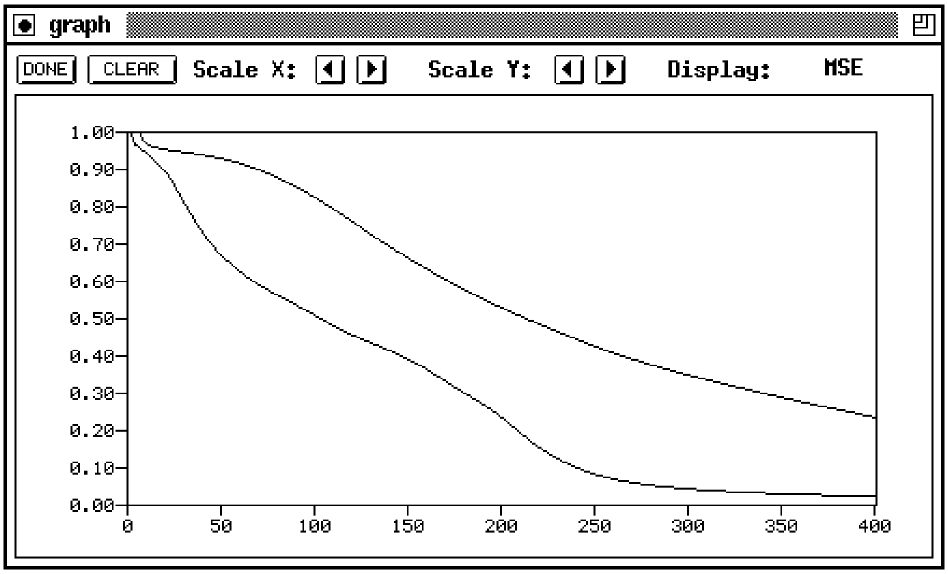

Figure 4.10: Graph window

Graph is only active after calling it. This means, the development of the error is only drawn as long as the window is not closed. The advantage of this implementation is, that the simulator is not slowed down as long as graph is closed[2]. If the window is iconified, graph remains active.

The error curve of the net is plotted until the net is initialized or a new net is loaded, in which case the cycle counter is reset to zero. The window, however, is not cleared until the clear button is pressed. This opens the possibility to compare several error curves in a single display (see also figure 4.10). The maximum number of curves, which can be displayed simultaneously is 25. If a 26^{th} curve is tried to be drawn, the confirmer appears with an error message.

When the curve reaches the right end of the window, an automatic rescale of the x-axis is performed. This way, the whole curve always remains visible.

In the top region of the graph window, seven buttons for handling the display are located:

`CLEAR`: Clears the screen of the graph window and sets the cycle counter to zero.

`DONE`: Closes the graph window and resets the cycle counter.

For both the x– and y–axis the following two buttons are available:

◀: Reduce scale in one direction.

[2]The loss of power by graph should be minimal.

▶: Enlarge scale in one direction.

`SSE`: Opens a popup menu to select the value to be plotted. Choices are `SSE`, `MSE`, and `SSE/out`, the SSE divided by the number of output units.

While the simulator is working all buttons are blocked.

The graph window can be resized by the mouse like every X-window. Changing the size of the window does not change the size of the scale.

When validation is turned on in the control panel two curves will be drawn simultaneously in the graph window, one for the training set and one for the validation set. On color terminals the validation error will be plotted as solid red line, on B/W terminals as dashed black line.

4.2.7 Weight Display

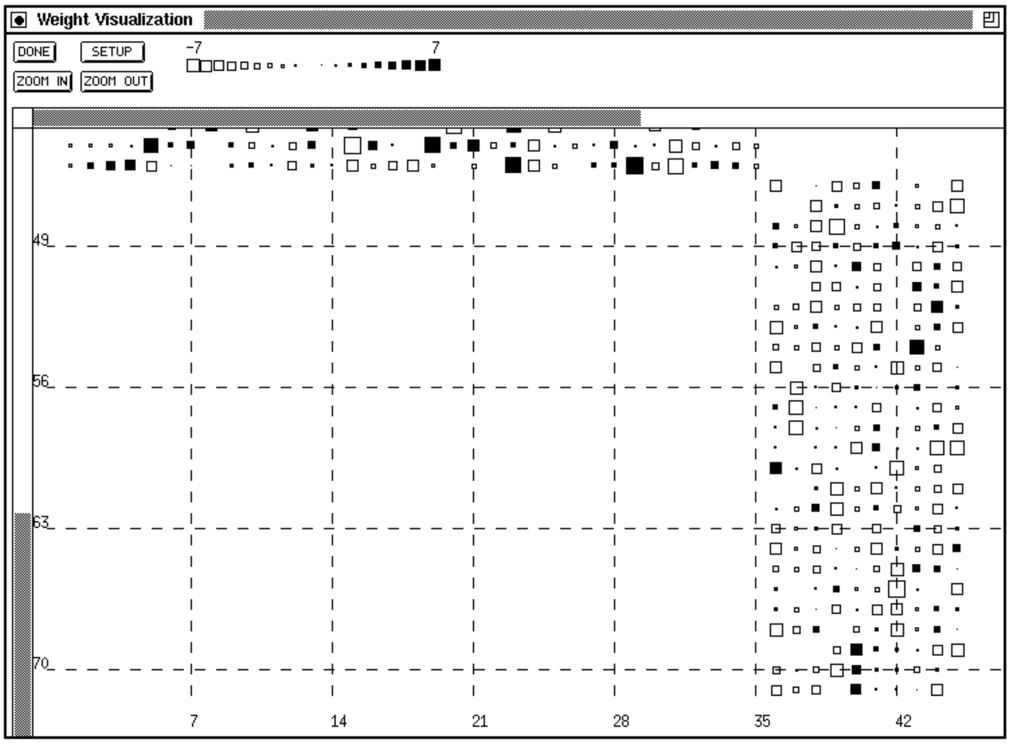

Figure 4.11: A typical Hinton diagram

The weight display window is a separate window specialized for displaying the weights of a network. It is called from the manager panel with the `WEIGHTS` button, or by typing Alt-w in any snns window. On black-and-white screens the weights are represented as squares with changing size in a Hinton diagram, while on color screens, fixed size squares with changing colors (WV-diagrams) are used. It can be used to analyze the weight distribution, or to observe the weight development during learning.

Initially the window has a size of 400x400 pixel.The weights are represented by 16^2 pixels on B/W and 5^2 pixels on color terminals. If the net is small, the square sizes are automatically enlarged to fill up the window. If the weights do not fit into the window, the scrollbars attached to the window allow scrolling over the display.

These settings may be changed by the user by pressing the ZOOM IN and ZOOM OUT buttons in the upper part of the window. ZOOM IN enlarges the weight square by one pixel on each side, while ZOOM OUT shrinks it.

The setup panel lets the user change the look of the display further. Here the width of the underlying grid can be changed. If the grid size is bigger than the number of connections in the network, no grid will be displayed. Also the color scale (resp. size scale for B/W) can be changed here. The initial settings correspond to the SNNS variables `max_weight` and `min_weight`.

In a Hinton diagram, the size of a square corresponds to the absolute size of the correlated link. A filled square represents positive, an square frame negative links. The maximum size of the squares is computed automatically, to allow an optimal use of the display. In a WV diagram color is used to code the value of a link. Here, a bright red is used for large negative values and a bright green is used for positive values. Intermediate numbers have a lighter color and the value zero is represented by white. A reference color scale is displayed in the top part of the window. The user also has the possibility to display the numerical value of the link by clicking any mouse button while the mouse pointer is on the square. A popup window then gives source and target unit of the current link as well as its weight.

For a better overall orientation the numbers of the units are printed all around the display and a grid with user definable size is used. In this numbering the units on top of the screen represent source units, while numbers to the left and right represent target units.

4.2.8 Projection Panel

The projection analysis tool allows to display how the output of one unit (e.g. a hidden or an output unit) depends on two input units. It thus realizes a projection to two input vector axes.

It can be called by clicking the `PROJECTION` button in the manager panel or by typing Alt-p in any SNNS window. The display of the projection panel is similar to the weights display, from which it is derived.

in the setup panel, two units must be specified, whose inputs are varied over the given input value range to give the X resp. Y coordinate of the projection display. The third unit to be specified is the one whose output value determines the color of the points with the given X and Y coordinate values. The range for the color coding can be specified as output range. For the most common logistic activation function this range is $[0, 1]$.

The use of the other buttons, ZOOM IN, ZOOM OUT and DONE are analogous to the weight display and should be obvious.

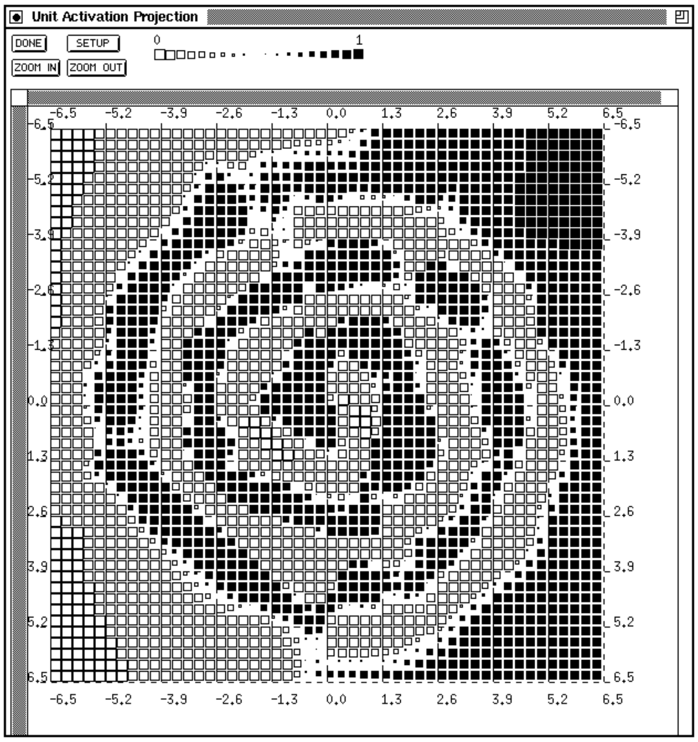

Figure 4.12: PROJECTION panel for the 2-spirals problem. Note that the input range of the X and Y units must be specified, not the output range. In contrast, the output range of the third unit specified determines the color. Here the following values have been used: X-axis: unit 1 value range -6.5 to 6.5; Y-axis: unit 2 value range -6.5 to 6.5; activation pattern: unit 38 (the only output unit of the network) value range 0 to 1

The projection tool is very instructive with the 2-spirals problem, the XOR problem or similar problems with two-dimensional input. Each hidden unit or output unit can be inspected and it can be determined, to which part of the input space the neuron is sensitive. Comparing different networks trained for such a problem by visualizing to which part of the input space they are sensitive gives insights about the internal representation of the networks and sometimes also about characteristics of the training algorithms used for training. A display of the projection panel is given in figure 4.12.

4.2.9 Print Panel

The print panel handles the Postscript output. A 2D-display can be associated with the printer. All setup options and values of this display will be printed. Color and encapsulated Postscript are also supported. The output device is either a printer or a file. If the output device is a printer, a '.ps'-file is generated and spooled in the /tmp directory. It has a unique name starting with the prefix 'snns'. The directory must be writable. When xgui terminates normally, all SNNS spool files are deleted.

The following fields can be set in the Printer Panel, which is shown in figure 4.13.

1. **File Name** resp. **Command Line**:
   If the output device is a file: the filename.
   If the output device is a printer: the command line to start the printer.
   The filename in the command line has to be '$1'.
2. **Destination**: Selects the output device. Toggles the above input line between File

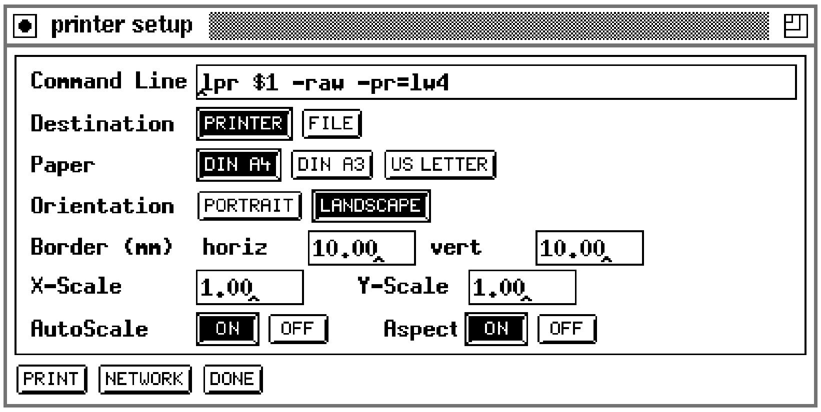

Figure 4.13: Printer panel

Name and Command Line.

3. **Paper**: Selects the paper format.
4. **Orientation**: Sets the orientation of the display on the paper. Can be 'portrait' or 'landscape'.
5. **Border (mm)**: Sets the size of the horizontal and vertical borders on the sheet in millimeters.
6. **AutoScale**: Scales the network to the largest size possible on the paper.
7. **Aspect**: If on, scaling in X and Y direction is done uniformly.
8. **X-Scale**: Scale factor in X direction. Valid only if AutoScale is 'OFF'.
9. **Y-Scale**: Scale factor in Y direction. Valid only if AutoScale is 'OFF'.

`DONE`: Cancels the printing and closes the panel.

`PRINT`: Starts printing.

`NETWORK`: Opens the network setup panel. This panel allows the specification of several options to control the way the network is printed.

The variables that can be set here include:

1. **x-min**, **y-min**, **x-max** and **y-max** describe the section to be printed.
2. **Unit size**:`FIXED`: All units have the same size.
   `VALUE`: The size of a unit depends on its value.
3. **Shape**: Sets the shape of the units.
4. **Text**:`SOLID`: The box around text overwrites the background color and the links.
   `TRANSPARENT`: No box around the text.
5. **Border**: A border is drawn around the network, if set to 'ON'.
6. **Color**: If set, the value is printed color coded.

7. **Fill Intens**: The fill intensity for units on monochrome printers.
8. **Display**: Selects the display to be printed.

4.2.10 Help Windows

An arbitrary number of help windows may be opened, each displaying a different part of the text. For a display of context sensitive help about the editor commands, the mouse must be in a display and the key `h` must be pressed. Then the last open help window appears with a short description.

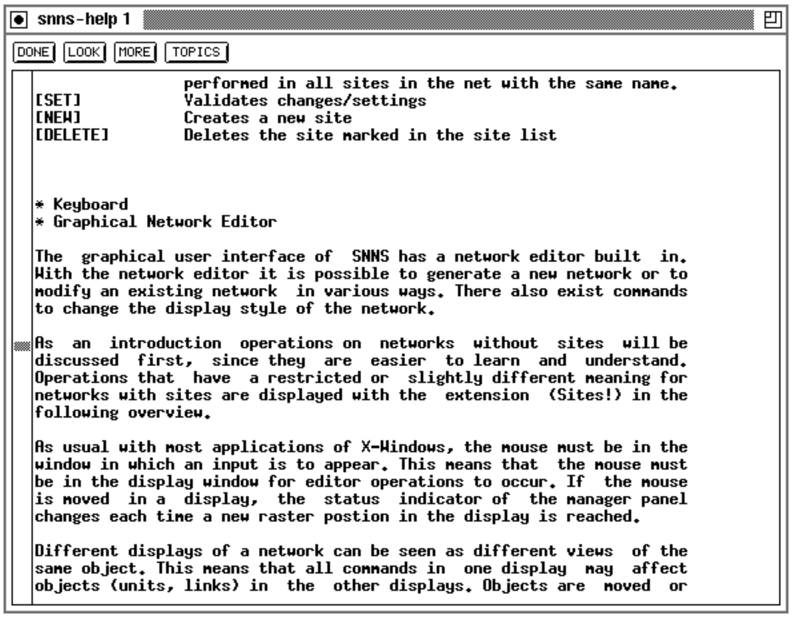

Figure 4.14: Help Window

A special feature is the possibility of searching a given string in the help text. For this, the search string is selected in the text window (e.g. by a double click).

1. `LOOK`: After clicking this button, SNNS looks for the first appearance of the marked string, starting at the beginning of the help document. If the string is found, the corresponding paragraph is displayed.
2. `MORE`: After clicking this button, SNNS looks for the first appearance of the marked string, starting at the position last visited by a call to the help function. If the text was scrolled afterwards, this position might not be on the display anymore.

Note: All help calls look for the first appearance of a certain string. These strings start with the sequence ASTERISK-BLANK (`* `), to assure the discovery of the appropriate text position. With this knowledge it is easy to modify the file `help.hdoc` to adapt it to special demands, like storing information about unit types or patterns. The best approach would be to list all relevant keywords at the end of the file under the headline "`* TOPICS`", so that the user can select this directory by a click to `TOPICS`.

4.2.11 Shell window

The window of the shell from which SNNS is invoked is used for the output of protocol messages.

These protocols include:

- Messages about the success or failure of the loading or saving of a file.
- Information about the settings of SNNS when the INFO button in the control panel is pressed.
- Error messages of the pattern file parser when the pattern file does not correspond to the required grammar.
- Learning error values (see below)
- Validation set error values

When learning is started, the error of the output units is reported on this window after each epoch, i.e. after the presentation of all patterns.

To save the window from being flooded on longer training runs, the maximum number of reported errors is limited to 10. Therefore, when 20 learning cycles are specified, the error gets printed only after every other cycle. This error report has the following form:

```
    Learning all patterns:
      epochs   : 100
      parameter:  0.80000
      #o-units : 26
      #patterns: 26

       epoch:         SSE            MSE            SSE/o-units

  Train  100:      57.78724       2.22259            2.22259
  Train   90:      24.67467       0.94903            0.94903
  Train   80:      23.73399       0.91285            0.91285
  Train   70:      22.40005       0.86154            0.86154
  Train   60:      20.42843       0.78571            0.78571
  Train   50:      18.30172       0.70391            0.70391
  Test    50:      25.34673       0.97487            0.97487
  Train   40:      16.57888       0.63765            0.63765
  Train   30:      14.84296       0.57088            0.57088
  Train   20:      12.97301       0.49896            0.49896
  Train   10:      11.22209       0.43162            0.43162
  Train    1:      10.03500       0.38596            0.38596
  Test     1:      11.13500       0.42696            0.42696
```

The first line reports whether all or only a single pattern is trained. The next lines give the number of specified cycles and the given learning parameters, followed by a brief setup description.

Then the 10-row-table of the learning progress is given. If validation is turned on this table is intermixed with the output of the validation. The first column specifies whether the displayed error is computed on the training or validation pattern set, “Test” is printed for the latter case. The second column gives the number of epochs still to be processed. The third column is the Sum Squared Error (SSE) of the learning function. It is computed with the following formula:

$$SSE = \sum_{p \in patterns} \sum_{j \in output} (t_{pj} - o_{pj})^2$$

where t_{pj} is the teaching output (desired output) of output neuron j on pattern p and o_{pj} is the actual output. The forth column is the Mean Squared Error (MSE), which is the SSE divided by the number of patterns. The fifth value finally gives the SSE divided by the number of output units.

The second and third values are equal if there are as many patterns as there are output units (e.g. the letters network), the first and third values are identical, if the network has only one output unit (e.g. the xor network).

If the training of the network is interrupted by pressing the $\boxed{\text{STOP}}$ button in the control panel, the values for the last completed training cycle are reported.

4.2.12 Confirmer

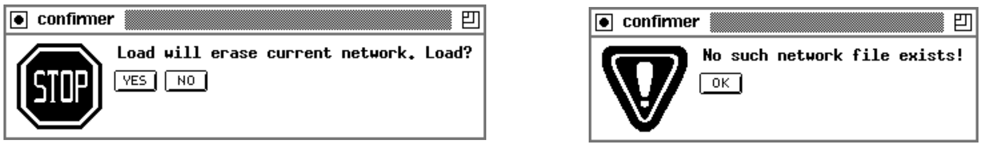

Figure 4.15: A normal confirmer and a message confirmer.

The confirmer is a window where the graphical user interface displays important information or requires the user to confirm destructive operations. The confirmer always appears in the middle of the screen and blocks XGUI until a button of the confirmer is clicked (see figure 4.15).

4.3 Parameters of the Learning Functions

The following learning parameters (from left to right) are used by the learning functions that are already built into SNNS:

- `Std_Backpropagation` ("Vanilla" Backpropagation), `BackpropBatch` and `TimeDelayBackprop`

  1. η: learning parameter, specifies the step width of the gradient descent.

     Typical values of η are $0.1 \ldots 1.0$. Some small examples actually train even faster with values above 1, like 2.0.

  2. d_{max}: the maximum difference $d_j = t_j - o_j$ between a teaching value t_j and an output o_j of an output unit which is tolerated, i.e. which is propagated back as $d_j = 0$. If values above 0.9 should be regarded as 1 and values below 0.1 as 0, then d_{max} should be set to 0.1. This prevents overtraining of the network.

     Typical values of d_{max} are 0, 0.1 or 0.2.

- `BackpropMomentum` (Backpropagation with momentum term and flat spot elimination):

  1. η: learning parameter, specifies the step width of the gradient descent.

     Typical values of η are $0.1 \ldots 1.0$. Some small examples actually train even faster with values above 1, like 2.0.

  2. μ: momentum term, specifies the amount of the old weight change (relative to 1) which is added to the current change.

     Typical values of μ are $0 \ldots 1.0$.

  3. c: flat spot elimination value, a constant value which is added to the derivative of the activation function to enable the network to pass flat spots of the error surface.

     Typical values of c are $0 \ldots 0.25$, most often 0.1 is used.

  4. d_{max}: the maximum difference $d_j = t_j - o_j$ between a teaching value t_j and an output o_j of an output unit which is tolerated, i.e. which is propagated back as $d_j = 0$. See above.

  The general formula for Backpropagation used here is

$$\begin{aligned} \Delta w_{ij}(t+1) &= \eta\ \delta_j\ o_i + \mu\ \Delta w_{ij}(t) \\ \delta_j &= \begin{cases} (f_j'(net_j) + c)(t_j - o_j) & \text{if unit } j \text{ is a output-unit} \\ (f_j'(net_j) + c)\sum_k \delta_k w_{jk} & \text{if unit } j \text{ is a hidden-unit} \end{cases} \end{aligned}$$

- `BackpropWeightDecay` (Backpropagation with Weight Decay)

  1. η: learning parameter, specifies the step width of the gradient descent.

     Typical values of η are $0.1 \ldots 1.0$. Some small examples actually train even faster with values above 1, like 2.0.

  2. d: weight decay term, specifies how much of the old weight value is subtracted after learning. Try values between 0.005 and 0.3.

  3. d_{max}: the maximum difference $d_j = t_j - o_j$ between a teaching value t_j and an output o_j of an output unit which is tolerated, i.e. which is propagated back as $d_j = 0$. See above.

- `BackpropThroughTime (BPTT)`,
  `BatchBackpropThroughTime (BBPTT)`:

  1. η: learning parameter, specifies the step width of the gradient descent.

     Typical values of η for BPTT and BBPTT are $0.005 \ldots 0.1$.

  2. μ: momentum term, specifies the amount of the old weight change (relative to 1) which is added to the current change.

     Typical values of μ are $0.0 \ldots 1.0$.

  3. backstep: the number of backprop steps back in time. BPTT stores a sequence of all unit activations while input patterns are applied. The activations are stored in a first-in-first-out queue for each unit. The largest backstep value supported is 10.

- `Quickprop`:

  1. η: learning parameter, specifies the step width of the gradient descent.

     Typical values of η for Quickprop are $0.1 \ldots 0.3$.

  2. μ: maximum growth parameter, specifies the maximum amount of weight change (relative to 1) which is added to the current change

     Typical values of μ are $1.75 \ldots 2.25$.

  3. ν: weight decay term to shrink the weights.

     Typical values of ν are 0.0001. Quickprop is rather sensitive to this parameter. It should not be set too large.

  4. d_{max}: the maximum difference $d_j = t_j - o_j$ between a teaching value t_j and an output o_j of an output unit which is tolerated, i.e. which is propagated back as $d_j = 0$. See above.

- `QuickpropThroughTime (QPTT)`:

  1. η: learning parameter, specifies the step width of the gradient descent.

     Typical values of η for QPTT are $0.005 \ldots 0.1$.

2. μ: maximum growth parameter, specifies the maximum amount of weight change (relative to 1) which is added to the current change

   Typical values of μ are $1.2 \ldots 1.75$.

3. ν: weight decay term to shrink the weights.

   Typical values of ν are $0.0005 \ldots 0.00005$.

4. backstep: the number of quickprop steps back in time. QPTT stores a sequence of all unit activations while input patterns are applied. The activations are stored in a first-in-first-out queue for each unit.

   The largest backstep value supported is 10.

- `Counterpropagation`:

  1. α: learning parameter of the Kohonen layer.

     Typical values of α for Counterpropagation are $0.1 \ldots 0.7$.

  2. β: learning parameter of the Grossberg layer.

     Typical values of β are $0 \ldots 1.0$.

  3. θ: threshold of a unit.

     We often use a value θ of 0.

- `Backpercolation 1`:

  1. λ: global error magnification. This is the factor in the formula $\epsilon = \lambda(t - o)$, where ϵ is the internal activation error of a unit, t is the teaching input and o the output of a unit.

     Typical values of λ are 1. Bigger values (up to 10) may also be used here.

  2. θ: If the error value ξ drops below this threshold value, the adaption according to the Backpercolation algorithm begins. ξ is defined as:

  $$\xi = \frac{1}{pN} \sum^{p} \sum^{N} \mid o - \phi \mid$$

  3. d_{max}: the maximum difference $d_j = t_j - o_j$ between a teaching value t_j and an output o_j of an output unit which is tolerated, i.e. which is propagated back as $d_j = 0$. See above.

- `Dynamic Learning Vector Quantization (DLVQ)`:

  1. η^+: learning rate, specifies the step width of the mean vector $\vec{\mu_A}$, which is nearest to a pattern $\vec{x_A}$, towards this pattern. Remember that $\vec{\mu_A}$ is moved only, if $\vec{x_A}$ is not assigned to the correct class w_A. A typical value is 0.03.

2. η^-: learning rate, specifies the step width of a mean vector $\vec{\mu_B}$, to which a pattern of class w_A is falsely assigned to, away from this pattern. A typical value is 0.03. Best results can be achieved, if the condition $\eta^+ = \eta^-$ is satisfied.

3. Number of cycles you want to train the net before additive mean vectors are calculated.

- `RadialBasisLearning`:

  1. *centers*: determines the learning rate η_1 used for the modification of center vectors.
  2. *bias (p)*: determines the learning rate η_2, used for the modification of the parameters p of the base function. p is stored as bias of the hidden units.
  3. *weights*: influences the training of all link weights that are leading to the output layer as well as the training of the bias of all output neurons.
  4. *delta max.*: If the actual error is smaller than the maximum allowed error (*delta max.*) the corresponding weights are not changed.
  5. *momentum*:influences the amount of the momentum–term during training.

- `RadialBasisLearning with Dynamic Decay Adjustment`:

  1. θ^+: positive threshold. To commit a new prototype, none of the existing RBFs of the correct class may have an activation above θ^+
  2. θ^-:negative threshold. During shrinking no RBF unit of a conflicting class is allowed to have an activation above θ^-.
  3. n: the maximum number of RBF units to be diplayed in one row. This item allows the user to control the appearance of the network on the screen and has no influence on the performance.

- `ART1`

  1. ρ: vigilance parameter. If the quotient of active F_1 units divided by the number of active F_0 units is below ρ, an ART reset is performed.

- `ART2`

  1. ρ: vigilance parameter. Specifies the minimal length of the error vector $\mathbf{r}$ (units r_i).
  2. a: Strength of the influence of the lower level in F_1 by the middle level.
  3. b: Strength of the influence of the middle level in F_1 by the upper level.
  4. c: Part of the length of vector $\mathbf{p}$ (units p_i) used to compute the error.
  5. Θ: Threshold for output function f of units x_i and q_i.

- `ARTMAP`
  1. $\overline{\rho^a}$: vigilance parameter for ART^a subnet. (quotient $\frac{|F_1^a|}{|F_0^a|}$)
  2. ρ^b: vigilance parameter for ART^b subnet. (quotient $\frac{|F_1^b|}{|F_0^b|}$)
  3. ρ: vigilance parameter for inter ART reset control. (quotient $\frac{|F^{ab}|}{|F_2^b|}$)
- `RPROP (resilient propagation)`
  1. $delta_0$: starting values for all $\triangle_{ij}$. Default value is 0.1.
  2. $delta_{max}$: the upper limit for the update values $\triangle_{ij}$.The default value of $\triangle_{max}$ is 50.0.
  3. α: the weight-decay determines the relationship between the output error and to reduction in the size of the weights.*Important:* Please note that the weight decay parameter α denotes the exponent, to allow comfortable input of very small weight-decay. A choice of the third learning parameter $\alpha = 4$ corresponds to a ratio of weight decay term to output error of $1 : 10000 (1 : 10^4)$.
- `Cascade Correlation (CC)` and
  `Recurrent Cascade Correlation (RCC)`

  CC and RCC are not learning functions themselves. They are meta algorithms to build and train optimal networks. However, they have a set of standard learning functions embedded. Here these functions require modified parameters. The embedded learning functions are:

  – Backpropagation (in CC or RCC):
    1. η_1: learning parameter, specifies the step width of gradient decent minimizing the net error.
    2. μ_1: momentum term, specifies the amount of the old weight change, which is added to the current change.
    3. c: flat spot elimination value, a constant value which is added to the derivative of the activation function to enable the network to pass flat spots on the error surface.
    4. η_2: learning parameter, specifies the step width of gradient ascent maximizing the covariance.
    5. μ_2: momentum term specifies the amount of the old weight change, which is added to the current change.

    The general formula for this learning function is:

    $$\Delta w_{ij}(t+1) \quad = \quad \eta S(t) + \mu \Delta w_{ij}(t-1)$$

    The slopes $\partial E / \partial w_{ij}$ and $-\partial C / \partial w_{ij}$ are abbreviated by S. This abbreviation is valid for all embedded functions. By changing the sign of the gradient value

$\partial C / \partial w_{ij}$, the same learning function can be used to maximize the covariance and to minimize the error.

– Rprop (in CC or RCC):

1. η_1^-: decreasing factor, specifies the factor by which the update-value Δ_{ij} is to be decreased when minimizing the net error. A typical value is 0.5.
2. η_1^+: increasing factor, specifies the factor by which the update-value Δ_{ij} is to be increased when minimizing the net error. A typical value is 1.2
3. not used.
4. η_2^-: decreasing factor, specifies the factor by which the update-value Δ_{ij} is to be decreased when maximizing the covariance. A typical value is 0.5.
5. η_2^+: increasing factor, specifies the factor by which the update-value Δ_{ij} is to be increased when maximizing the covariance. A typical value is 1.2

The weight change is computed by:

$$\Delta w_{ij}(t) = \begin{cases} -\Delta_{ij}(t-1)\eta^-, & \text{if } S(t)S(t-1) < 0 \\ -\Delta_{ij}(t-1)\eta^+, & \text{if } S(t) > 0 \text{ and } S(t-1) > 0 \\ \Delta_{ij}(t-1)\eta^+, & \text{if } S(t) < 0 \text{ and } S(t-1) < 0 \\ 0, & \text{else} \end{cases}$$

where $\Delta_{ij}(t)$ is defined as follows: $\Delta_{ij}(t) = \Delta_{ij}(t-1)\eta^{+/-}$. Furthermore, the condition $0 < \eta^- < 1 < \eta^+$ should not be violated.

– Quickprop (in CC or RCC):

1. η_1: learning parameter, specifies the step width of the gradient descent when minimizing the net error. A typical value is 0.0001
2. μ_1: maximum growth parameter, realizes a kind of dynamic momentum term. A typical value is 2.0.
3. ν: weight decay term to shrink the weights. A typical value is <= 0.0001.
4. η_2: learning parameter, specifies the step width of the gradient ascent when maximizing the covariance. A typical value is 0.0007
5. μ_2: maximum growth parameter, realizes a kind of dynamic momentum term. A typical value is 2.0.

The formula used is:

$$\Delta w_{ij}(t) = \begin{cases} \eta S(t), & \text{if } \Delta w_{ij}(t-1) = 0 \\ \frac{S(t)}{S(t-1)-S(t)}\Delta w_{ij}(t-1), & \text{if } \Delta w_{ij}(t-1) \neq 0 \text{ and } \frac{S(t)}{S(t-1)-S(t)} < \mu \\ \mu \Delta w_{ij}(t-1), & \text{else} \end{cases}$$

- `Kohonen`

1. *h(0):* Adaptation height. The initial adaptation height can vary between 0 and 1. It determines the overall adaptation strength.

2. *r(0):* Adaptation radius. The initial adaptation radius $r(0)$ is the radius of the neighborhood of the winning unit. All units within this radius are adapted. Values should range between 1 and the size of the map.

3. *mult_H:* Decrease factor. The adaptation height decreases monotonically after the presentation of every learning pattern. This decrease is controlled by the decrease factor mult_H: $h(t+1) := h(t) \cdot mult\_H$

4. *mult_R:* Decrease factor. The adaptation radius also decreases monotonically after the presentation of every learning pattern. This second decrease is controlled by the decrease factor mult_R: $r(t+1) := r(t) * mult\_R$

5. *h:* Horizontal size. Since the internal representation of a network doesn't allow to determine the 2-dimensional layout of the grid, the horizontal size in units must be provided for the learning function. It is the same value as used for the creation of the network.

- `RM_delta (Rumelhart and McClelland's delta rule)`

1. *n:* learning parameter, specifies the step width of the gradient descent. In [RM86] Rumelhart and McClelland use 0.01, although values less than 0.03 are generally acceptable.

2. *Ncycles:* number of update cycles, specifies how many times a pattern is propagated through the network before the learning rule is applied. This parameter must be large enough so that the network is relatively stable after the set number of propagations. A value of 50 is recommended as a baseline. Increasing the value of this parameter increases the accuracy of the network but at a cost of processing time. Larger networks will probably require a higher setting of Ncycles.

NOTE: With this learning rule the update function `RM_Synchronous` has to be used which needs as update parameter the number of iterations!

- `Hebbian Learning`

1. *n:* learning parameter, specifies the step width of the gradient descent. Values less than (1 / number of nodes) are recommended.

2. *Wmax:* maximum weight strength, specifies the maximum absolute value of weight allowed in the network. A value of 1.0 is recommended, although this should be lowered if the network experiences explosive growth in the weights and activations. Larger networks will require lower values of Wmax.

3. *count:* number of times the network is updated before calculating the error.

NOTE: With this learning rule the update function `RM_Synchronous` has to be used which needs as update parameter the number of iterations!

4.4 Update Functions

Why is an update mode important? It is necessary to visit the neurons of a net in a specific sequential order to perform operations on them. This order depends on the topology of the net and greatly influences the outcome of a propagation cycle. For each net with its own characteristics, it is very important to choose the update function associated with the net in order to get the desired behavior of the neural network. If a wrong update function is given, SNNS will display a error message on your screen. Click `OPTIONS` in the control panel to select an update function.

The following update functions are available for the various network types:

ART1_stable	for ART1 networks
ART1_synchronous	for ART1 networks
ART2_Stable	for ART2 networks
ART2_Synchronous	for ART2 networks
ARTMAP_Stable	for ARTMAP networks
ARTMAP_Synchronous	for ARTMAP networks
Auto_Synchronous	for Autoassociative Memory networks
BAM_Order	for Bidirectional Associative Memory networks
BBTT_Order	for Backpropagation-Through-Time networks
CC_Order	for Cascade- Correlation networks
CounterPropagation	for Counterpropagation networks
Dynamic_LVQ	for Dynamic-Learning-Vector-Quantization networks
Hopfield_Fixed_Act	for Hopfield networks
Hopfield_Synchronous	for Hopfield networks
JE_Order	for Jordan or Elman network
JE_Special	for Jordan or Elman network
Kohonen_Oder	for Self-Organizing Maps (SOMS)
RCC_Order	for reccurrent-Cascade-Correlation networks
Random_Order	for any network
Random_Permutation	for any network
Serial_Order	for any network
Synchronous_Order	for any network
TimeDelay_Order	for Time-Delay networks
Topological_Order	for any network

All these functions receive their input from the five update parameter fields in the control panel. See figure 4.5

The following parameters are required for ART, Hopfield, and Autoassociative networks. The description of the update functions will indicate which parameter is needed.

$$\begin{aligned}
\rho\ &= \text{vigilance parameter with } (0 \leq \rho \leq 1.) : \text{field1} \\
\rho^a &= \text{initial vigilance parameter for the ART}^a \text{ part of the net} \\
&\quad\ \text{with } (0 < \rho^a < 1) : \text{field1} \\
\rho^b &= \text{vigilance parameter for ART}^b \text{ with } (0 < \rho^b < 1) : \text{field2}
\end{aligned}$$

ρ^c = Inter-ART-Reset control with $(0 < \rho^b < 1)$: field3
a = strength of the influence of the lower level in F1 by the middle level with $(a > 0)$: field2
b = strength of the influence of the middle level in F1 by the upper level with $(b > 0)$: field3
c = part of the length of vector p with (0 < c < 1) : field4
Θ = Kind of threshold with $(0 \leq \Theta < 1.)$: field5
c = number of units : field1
n = iteration parameter : field1
x = number of selected neurons

Field1..Field5 are the positions in the control panel. For a more detailed description of ART-parameters see section 9.12: ART Models in SNNS

Now here is a description of the steps the various update functions perform, and of the way in which they differ.

ART1_Stable

The `ART1_Stable` update function updates the neurons activation and output values until a stable state is reached. In one propagation step the activation of all non-input units is calculated and then the calculation of the output of all neuron follows. The state is considered stable if the 'classifiable' or the 'not classifiable' neuron is selected. 'Classifiable' means that the input vector (pattern) is recognized by the net. 'Not classifiable' means that that there is no neuron in the recognition layer which would fit the input pattern. The required parameter is ρ in field1.

ART1_Synchronous

The algorithm of the `ART1_Synchronous` update function is the ART1 equivalent to the algorithm of the Synchronous_Order function. The only difference is that the winner of the ART1 recognition layer is identified. The required parameter is ρ in field1.

ART2_Stable

The first task of this algorithm is to initialize the activation of all units. This is necessary each time a new pattern is loaded to the network. The ART2 net is initialized for a new pattern now. The output and activation will be updated with synchronous propagations until a stable state is reached. One synchronous propagation cycle means that each neuron calculate its output and then its new activation. The required parameters are ρ, a, b, c, Θ in field1, field2, field3, field4, field5 respectively.

ART2_Synchronous

This function is the ART2 equivalent to the `Synchronous_Order` function. The only difference is that additionally the winner neuron of the ART1 recognition layer is calculated. The required parameters are ρ, a, b, c, Θ in field1, field2, field3, field4, field5 respectively.

ARTMAP_Stable

`ARTMAP_Stable` updates all units until a stable state is reached. The state is considered stable if the classified or unclassified unit is 'on'. All neurons compute their output and activation in one propagation step. The propagation step continues until the stable state is reached. The required parameters are ρ^a, ρ^b, ρ^c in field1, field2, field3 respectively.

ARTMAP_Synchronous

The first step is to calculate the output value of the input units (input units of ARTa, ARTb). Now a complete propagation step takes place, i.e. all units calculate their output and activation value. The search for two recognition neuron with highest activation follows. The search takes place in both ARTa and ARTb. The required parameters are ρ^a, ρ^b, ρ^c in field1, field2, field3 respectively.

Auto_Synchronous

First the `Auto_Synchronous` function calculates the activation of all neurons. The next step is to calculate the output of all units. The two steps will be repeated n times. For the iteration parameter n, which has to be provided in field1, a value of 50 has shown to be very suitable.

BAM_Order

The first step of this update function is to search for the first hidden unit of the network. The current output is saved and a new output is calculated for all neurons of the hidden and output layer. Once this is accomplished the next progression of the hidden and output units starts. Now for each neuron of the hidden and output layer the new output is saved and the old saved output is restored. With this older output the activation of all hidden and output neurons is calculated. After this task is accomplished the new saved output value of all hidden and output neurons is restored.

BBTT_Order

The `BBTT_Order` algorithm performs an update on a recurrent network. The recurrent net can be transformed into a regular feedforward net with an input, multiple hidden and output layer. At the beginning the update procedure checks if there is a zero-input pattern in the input layer. Suppose there is such a pattern, then the so called `i_act` value buffer is set to 0 for all neurons. In this case `i_act` can be seen as a buffer for the output value of the hidden and output neurons. The next step is to copy the `i_act` value to the output of all hidden and output neurons. The new activation of the hidden and output units will be calculated. Now the new output for every neuron in the hidden and output layer will be computed and stored in `i_act`.

CC_Order

The `CC_Order` update function propagates a pattern through the net. This means all neurons calculate their new activation and output in a topological order. The `CC_Order` update function also handles the special units which represent the candidate units.

CounterPropagation

The `CounterPropagation` update algorithm updates a net that consists of a input, hidden and output layer. In this case the hidden layer is called the Kohonen layer and the output layer is called the Grossberg layer. At the beginning of the algorithm the output of the input neurons is equal to the input vector. The input vector is normalized to the length of one. Now the progression of the Kohonen layer starts. This means that a neuron with the highest net input is identified. The activation of this winner neuron is set to 1. The activation of all other neurons in this layer is set to 0. Now the output of all output neurons is calculated. There is only one neuron of the hidden layer with the activation and the output set to 1. This and the fact that the activation and the output of all output neurons is the weighted sum on the output of the hidden neurons implies that the output of the output neurons is the weight of the link between the winner neuron and the output neurons. This update function makes sense only in combination with the CPN learning function.

Dynamic_LVQ

This update algorithm initializes the output and activation value of all input neurons with the input vector. Now the progression of the hidden neurons begins. First the activation and output of each of the hidden neurons is initialized with 0 and the new activation will be calculated. The hidden neuron with the highest activation will be identified. Note that the activation of this winner unit has to be > -1. The class which the input pattern belongs to will be propagated to the output neuron and stored as the neurons activation. This update function is sensible only in combination with the DLVQ learning function.

Hopfield_Fixed_Act

This update function selects x neurons with the highest net-inputs and associates the activation value of those units with 1. The activation value of all other units is associated with 0. Afterwards the output value of all neurons will be calculated. The required parameter is x in field1.

Hopfield_Synchronous

This update function calculates the output of all neurons first. This has to be done in order to propagate the pattern which is represented by the input vector. The activation update of all neurons which are not input neurons follows. The next step is to calculate the output value of those units. The input units are handled next. The activation of the input neurons is calculated and the next progression updates the output of all input units.

JE_Order

This update function propagates a pattern from the input layer to the first hidden layer, then to the second hidden layer, etc. and finally to the output layer. After this follows a synchronous update of all context units. This function is makes sense only for JE-networks.

JE_Special

Using the update function `JE_Special`, input patterns will be generated dynamically. Let n be the number of input units and m the number of output units of the network. `JE_Special` generates the new input vector with the output of the last $n - m$ input units and the outputs of the m output units. The usage of this update function requires $n > m$. The propagation of the newly generated pattern is done like using `JE_Update`. The number of the actual pattern in the control panel has no meaning for the input pattern when using `JE_Special`. This update function is used to determine the prediction capabilities of a trained network.

Kohonen_Order

The `Kohonen_Order` function propagates neurons in a topological order. There are 2 propagation steps. The first step all input units are propagated, which means that the output of all neurons is calculated. The second step consists of the propagation of all hidden units. This propagation step calculates all hidden neuron's activation and output. Please note that the activation and output are normally not required for the Kohonen algorithm. The activation and output values are used for display and evaluation reasons internally. The `Act_Euclid` activation function for example, copies the Euclidian distance of the unit from the training pattern to the units activation.

RCC_Order

This function is similar to the `CC_Order` except it takes special care of the recurrent links. Use this function only for RCC-networks.

Random_Order

The `Random_Order` update function selects a neuron and calculates its activation and output value. The selection process is absolutely random and will be repeated n times. The parameter n is the number of existing neurons. One specific neuron can be selected more than one time while other neurons may be left out. This kind of update function is rarely used and is just a theoretical base to prove the stability of Hopfield nets.

Random_Permutation

This update function is similar to the `Random_Order` function. The only difference is that a random permutation of all neurons is used to select the order of the units. This guarantees that each neuron will be selected exactly once to calculate the output and activation value. This procedure has two big disadvantages. The first disadvantage is that the computation of the permutation is very time consuming and the second disadvantage is that it takes a long time until a stable output vector has been established.

Serial_Order

The `Serial_Order` update function calculates the activation and output value for each unit. The progression of the neurons is serial which means the computation process starts at the first unit and proceeds to the last one.

Synchronous_Order

With the synchronous update function all neurons change their value at the same time. All neurons calculate their activation in one single step. The output of all neurons will be calculated after the activation step. The difference to the `serial_order` update function is that the calculation of the output and activation value requires two progressions of all neurons. This kind of propagation is very useful for distributed systems (SIMD).

TimeDelay_Order

The update function TimeDelay_Order is used to propagate patters through a time delay network. Its behavior is analogous to the Topological_Order functions with recognition of logical links.

Topological_Order

This mode is the most favorable mode for feedforward nets. The neurons calculate their new activation in a topological order. The topological order is given by the net-topology. This means that the first processed layer is the input layer. The next processed layer is the first hidden layer and the last layer is the output layer. A learning cycle is defined as a pass through all neurons of the net. Shortcut-connections are allowed.

4.5 Initialization Functions

In order to work with various neural network models and learning algorithms, different initialization functions that initialize the components of a net are required. Backpropagation, for example, will not work properly if all weights are initialized to the same value.

To select an initialization function, one must click `SEL. FUNC` in the `INIT` line of the control panel.

The following initialization functions are available:

ART1_Weights	for ART1 networks
ART2_Weights	for ART2 networks
ARTMAP_Weights	for ARTMAP networks
CC_Weights	for Cascade Correlation networks
ClippHebb	for Associative Memory networks
CPN_Rand_Pat	for Counterpropagation
CPN_Weights_v3.2	for Counterpropagation
CPN_Weights_v3.3	for Counterpropagation
DLVQ_Weights	for Dynamic Learning Vector Quantization
Hebb	for Associative Memory networks
Hebb_Fixed_Act	for Associative Memory networks
JE_Weights	for Jordan or Elman networks
Kohonen_Rand_Pat	for Self-Organizing Maps (SOMS)
Kohonen_Const	for Self-Organizing Maps (SOMS)
Kohonen_Weights_v3.2	for Self-Organizing Maps (SOMS)
Pseudoinv	for Associative Memory networks
Randomize_Weights	for any network, except the ART-family
Random_Weights_Perc	for Backpercolation
RBF_Weights	for Radial Basis Functions (RBFs)
RBF_Weights_Kohonen	for Radial Basis Functions (RBFs)
RBF_Weights_Redo	for Radial Basis Functions (RBFs)
RCC_Weights	for recurrent Cascade-Correlation networks
RM_Random_Weights	for Autoassociative Memory Networks

All these functions receive their input from the five init parameter fields in the control panel. See figure 4.5

Here is a short description of the different initialization functions:

ART1_Weights

ART1_Weights is responsible to set the initial values of the trainable links in an ART1 network. These links are the ones from F_1 to F_2 and the ones from F_2 to F_1 respectively. For more details see chapter 9.12.1.2.

ART2_Weights

For an ART2 network the weights of the top-down-links ($F_2 \rightarrow F_1$ links) are set to 0.0 according to the theory ([CG87b]). The choice of the initial bottom-up-weights is described in chapter 9.12.2.2.

ARTMAP_Weights

The trainable weights of an ARTMAP network are primarily the ones of the two ART1 networks ART^a and ART^b, therefore the initialization process is similar. For more details see chapter 9.12.1.2 and chapter 9.12.2.2.

CC_Weights

`CC_Weights` calls the `Randomize_Weights` function. See `Randomize_Weights`.

ClippHebb

The `ClippHebb` algorithm is almost the same as the `Hebb` algorithm, the only difference is that all weights can only be set to 1 and 0. After the activation for the neurons is calculated, all weights > 1 will be set to 1. As mentioned in 4.5 the `ClippHebb` algorithm is a learning algorithm.

CPN_Rand_Pat

This Counterpropagation initialization function initializes all weight vectors of the Kohonen layer with random input patterns from the training set. This guarantees that the Kohonen layer has no dead neurons.

The weights of the Grossberg layer are all initialized to 1.

CPN_Weights_v3.2

This function generates random points in an n-dimensional hypercube and later projects them onto the surface of an n-dimensional unit hypersphere or onto one of its main diagonal sectors (main diagonal quadrant for $n = 2$, octant for $n = 3$, ...).

First the interval, from which the Kohonen weights for the initialization tasks are selected, is determined. Depending upon the initialization parameters, which have to be provided in field1 and field2, the interval may be $[-1\,;+1]$, $[0\,;+1]$, or $[-1\,;0]$.

Every component w_{ij} of every Kohonen layer neuron j is then assigned a random value from the above interval, yielding weight vectors $\vec{w_j}$, which are random points within an n-dimensional hypercube.

The length of each vector $\vec{w}_j$ is then normalized to 1.

All weights of the neurons of the Grossberg layer are set to 1.

Note that this initialization function does NOT produce weight vectors with equal point density on the hypersphere, because with increasing dimension of the hypercube in which the random dots are generated, many more points are originally in the corners of the hypercube than in the interior of the inscribed hypersphere.

CPN_Weights_v3.3

This function generates random points in an n-dimensional cube, throws out all vectors with length > 1 and projects the remaining onto the surface of an n-dimensional unit hypersphere or onto one of its main diagonal sectors (main diagonal quadrant for $n = 2$, octant for $n = 3$, ...).

First the interval, from which the Kohonen weights for the initialization tasks are selected, is determined. Depending upon the initialization parameters, which have to be provided in field1 and field2, the interval may be $[-1\,;+1]$, $[0\,;+1]$, or $[-1\,;0]$.

Every component w_{ij} of every Kohonen layer neuron j is then assigned a random value from the above interval, yielding weight vectors $\vec{w}_j$, which are random points within an n-dimensional hypercube. If the weight vector $\vec{w}_j$ thus generated is outside the unit hypersphere or hypersphere sector, a new random vector is generated until eventually one is inside the hypersphere or hypersphere sector. Finally, the length of each vector $\vec{w}_j$ is normalized to 1.

The Grossberg layer weight vector components are all set to 1.

Note that this initialization function DOES produce weight vectors with equal point density on the hypersphere. However, the fraction of points from the hypercube which are inside the enscribed hypersphere decreases exponentially with increasing vector dimension, thus exponentially increasing the time to perform the initialization. This method is thus only suitable for input dimensions up to 12 - 15. (read Hecht-Nielsen: Neurocomputing, chapter 2.4, pp. 41 ff. for an interesting discussion on n-dim. geometry).

DLVQ_Weights

`DLVQ_Weights` calls the `Randomize_Weights` function. See `Randomize_Weights`.

Hebb

This procedure is similar to the Hebbian Learning Rule with a learning rate of 1. Additionally the bias of all input and output neurons is set with the parameters p1 and p2 which have to be provided in field1 and field2. Please note that the `Hebb`, `ClippHebb`, `HopFixAct` and `PseudoInv` initialization functions are actually learning functions. The reason why those functions are called initialization functions, is the fact that there is no true training because all weights will be calculated directly. In case the values of the parameters p1 and p2 are 1 and -1 the bias of the input and output neurons will be set to ld (n) and ld (k). Where n is the number of input neurons and k is the number of output neurons. These settings are also the default settings for p1 and p2. In any other case the p1 and p2 represent the bias of the input and output neurons without any modification.

Hebb_FixAct

This rule is necessary in order to do 'one-step-recall' simulations. For more informations see [Ama89]. For the calculation the bias following facts are assumed:

- The implemented net is an autoassociative net
  The neurons of an autoassociative net have to be input and output neurons at the same time. A Hopfield network would be an example for such a net.
- Fixed number of 1s
  The patterns which are to be saved have a fixed number of 1s.

The parameter h1 and h2 are required. Where h1 is the number of ones per pattern and h2 is the probable degree of distortion in percent. The parameters have to be inserted in field1 and field2. This initialization function should be used only in connection with the `Hopfield_Fixed_Act` update function. As mentioned in section 4.5 the `Hebb_FixAct` algorithm is a learning algorithm.

JE_Weights

This network consists of two types of neurons. The regular neurons and the so called context neurons. In such networks all links leading to context units are considered recurrent links. The initialization function `JE_Weights` requires the specification of five parameters:

- α, β : The weights of the forward connections are randomly chosen from the interval $[\alpha\,;\beta]$. α, β have to be provided in field1 and field2 of the init panel.
- λ : Weights of self recurrent links from context units to themselves. Simple Elman networks use $\lambda = 0$. λ has to be provided in field3 of the init panel.
- γ : Weights of other recurrent links to context units. This value is often set to 1.0. γ has to be provided in field4 of the init panel.
- ψ : Initial activation of all context units. ψ has to be provided in field5 of the init panel.

Note that it is required that $\alpha > \beta$. If this is not the case, an error message will appear on the screen. The context units will be initialized as described above. For all other neurons the bias and all weights will be randomly chosen from the interval $[\alpha\,;\beta]$.

Kohonen_Weights_v3.2

This initialization function is identical to CPN_Weights_v3.2 except that it only initializes the Kohonen layer, because there is no second Grossberg layer as in Counterpropagation.

Kohonen_Const

Each component w_{ij} of each Kohonen weight vector w_j is set to the value of $\frac{1}{\sqrt{n}}$, thus yielding all identical weight vectors w_j of length 1. This is no problem, because the Kohonen algorithm will quickly pull weight vectors away from this central dot and move them into the proper direction.

Kohonen_Rand_Pat

This initialization function initializes all weight vectors of the single Kohonen layer with random input patterns from the training set. This guarantees that the Kohonen layer initially has no dead neurons.

PseudoInv

The `PseudoInv` initialization function computes all weights with the help of the pseudo inverse weight matrix which is calculated with the algorithm of Greville. The formula for the weight calculation is: W = QS^+. Where S^+ is the 'Pseudoinverse' of the input vectors, Q are the output vectors and W are the desired weights of the net. The bias is not set and there are no parameters necessary. Please note that the calculated weights are usually odd. As mentioned in 4.5 the `PseudoInv` algorithm is a learning algorithm.

Randomize_Weights

This function initializes all weights and the bias with distributed random values. The values are chosen from the interval $[\alpha\,;\beta]$. α and β have to be provided in field1 and field2 of the init panel. It is required that $\alpha > \beta$.

Random_Weights_Perc

The first task of this function is to calculate the number of incoming links of a unit. Once this is accomplished, the range of possible weight-values will be determined. The range will be calculated with the α and β parameters, which have to be provided with the help of the init panel in field1 and field2. If $\alpha = \beta$ all weights and the bias will be set to the value of the α parameter. If $\alpha <> \beta$ the links of all neurons will be initialized with random values selected from the interval $[\alpha\,;\beta]$ and divided by the number of incoming links of every neuron. The bias will also be set to zero.

RBF_Weights

This procedure first selects evenly distributed centers $\vec{t_j}$ from the loaded training patterns and assigns them to the links between input and hidden layer. Subsequently the bias of all neurons (parameter p) inside the hidden layer is set to a value determined by the user and finally the links between hidden and output layer are computed. For more details see chapter 9.10.2.2.

RBF_Weights_Kohonen

Using the self-organizing method of Kohonen feature maps, appropriate centers are generated on base of the teaching patterns. The computed centers are copied into the corresponding links. No other links and bias are changed. For more details see chapter 9.10.2.2.

RBF_Weights_Redo

In contrast to the preceding procedure only the links between hidden and output layer are computed. All other links and bias remain unchanged. For more details see chapter 9.10.2.2.

RCC_Weights

RCC_Weights calls the `Randomize_Weights` function. See `Randomize_Weights`.

RM_Random_Weights

The `RM_Random_Weights` function initializes the bias and all weights of all units which are not input-units with a random value. This value is selected from the interval $[\alpha\,;\beta]$. α and β have to be provided in field1 and field2 of the init panel. $\alpha > \beta$ has to hold.

4.6 Creating and Editing Unit Prototypes and Sites

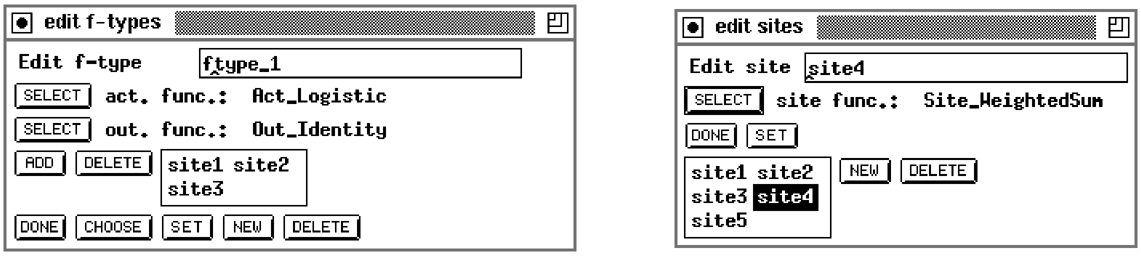

Figure 4.16: Edit panels for unit prototypes (f-types) and sites

Figure 4.16 shows the panels to edit unit prototypes (f-types) and sites. The change of the f-type is performed on all units of that type. Therefore, the functionality of all units assigned to an f-type can easily be changed. The elements in the panel have the following meaning:

- `SELECT`: Selects of the activation and output function.
- `CHOOSE`: Chooses the f-type to be changed.
- `SET`: Makes the settings/changes permanent. Changes in the site list are not set (see below).
- `NEW`, `DELETE`: Creates or deletes an f-type.
- `ADD`, `DELETE`: F-types also specify the sites of a unit. Therefore these two buttons are necessary to add/delete a site in the site list.

Note: The number and the selection of sites can not be changed after the creation of an f-type.

The elements in the edit panel for sites are almost identical. A site is selected for change by clicking at it in the site list.

- `SELECT`: Selects the new site function. The change is performed in all sites in the net with the same name.
- `SET`: Validates changes/settings.
- `NEW`: Creates a new site.
- `DELETE`: Deletes the site marked in the site list.

Chapter 5

Handling Patterns with SNNS

The general case of using patterns with a neural network is an exact fit of the patterns onto the network. The set of activations of all input units is called input pattern, the set of activations of all output units is called output pattern. The input pattern and its corresponding output pattern is simply called a pattern. This definition implies that all patterns for a particular network have the same size. These patterns will be called *regular* or *fixed sized*.

SNNS also offers another, much more flexible type of pattern. These patterns will be called *variable sized*. Here, the patterns are usually larger than the input/output layers of the network. To train and recall these patterns small portions (subsequently called *subpatterns*) are systematically cut out from the large pattern and propagated through the net, one at a time. Only the smaller subpatterns do have to have the fixed size fitting the network. The pattern itself may have an arbitrary size and different patterns within one pattern set may have differing sizes. The number of variable dimensions is also variable. Example applications for one and two variable dimensions include time series patterns for TDNNs and picture patterns.

Both of these types of patterns are loaded into SNNS from the same kind of pattern file. For a detailed description of the structure of this file see sections 5.2 and 5.3. The grammar is given in appendix A.4

5.1 Handling Pattern Sets

Although activations can be propagated through the network without patterns defined, learning can be performed only with patterns present. A set of patterns belonging to the same task is called a pattern set. Normally there are two dedicated pattern sets when dealing with a neural network. One for training the network (training pattern set), and one for testing purposes to see what the network has learned (test pattern set). In SNNS both of these (and more) can be kept in the simulator at the same time. They are loaded with the file browser (see chapter 4.2.2). The pattern set loaded last is made the current pattern set. All actions performed with the simulator refer only to, and affect only the current pattern set. To switch between pattern sets press the button $\boxed{\text{USE}}$ in the control panel (see figure 4.5 on page 30). It opens up a list of loaded pattern sets from which a new one can be selected. The name of the current pattern set is displayed to the right of the button. The name equals the name body of the loaded pattern file. If no pattern set is loaded, "Patternfile ?" is given as indication that no associated pattern file is defined.

Loaded pattern sets can be removed from main memory with the DELETE button in the control panel. Just like the USE button it opens a list of loaded pattern sets, from which any set can be deleted. When a pattern set is deleted, the corresponding memory is freed, and again available for other uses. This is especially important with larger pattern sets, where memory might get scarce.

5.2 Fixed Size Patterns

When using fixed size patterns, the number of input and output values has to match the number of input and output units in the network respectively for training purposes. Patterns without output activations can be defined for networks without output units (e.g. ART networks), or for test/recall purposes for networks with output units. It is possible, for example, to load two sets of patterns into SNNS: A training pattern set with output values for the training of the network, and a test pattern set without output values for recall. The switch between the pattern sets is performed with the USE button as described above.

Pattern definition files prior to V3.0 require output activations. Networks or patterns without output are not possible[1]

Note: All Pattern definition files prior to V3.2 now correspond to the type fixed size!

5.3 Variable Size Patterns

Variable patterns are much more difficult to define and handle. Example applications for variable pattern set include TDNN patterns for one variable dimension and picture processing for two variable dimensions. The SNNS pattern definition is very flexible and allows a great degree of freedom. Unfortunately this also renders the writing of correct pattern files more difficult and promotes mistakes.

To make the user acquainted with the pattern file format we describe the format with the help of an example pattern file. The beginning of the pattern file describing a bitmap picture is given below. For easier reference, line numbers have been added on the left.

```
0001  SNNS pattern definition file V3.2
0002  generated at Fri Aug  3 00:00:44 1999
0003
0004  No. of patterns : 10
0005  No. of input units : 1
0006  No. of output units : 1
0007  No. of variable input dimensions : 2
```

[1]SNNSv4.0 reads all pattern file formats, but writes only the new, flexible format. This way SNNS itself can be used as a conversion utility.

```
0008  Maximum input dimensions : [ 200 200 ]
0009  No. of variable output dimensions : 2
0010  Maximum output dimensions : [ 200 200 ]
0011
0012  # Input pattern 1: pic1
0013  [ 200 190 ]
0014  1 1 1 0 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1 0 1 1 0 0 1 1 0 1
      .
      .
      .
0214  1 1 1 0 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1 0 1 1 0 0 1 1 0 1
0215  # Output pattern 1: pic1
0216  [ 200 190 ]
0217  1 1 1 0 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1 0 1 1 0 0 1 1 0 1
      .
      .
      .
```

Some of the comments identifying parameter names make no sense when the file describes a variable pattern. They are kept, however, for reasons of compatibility with the regular fixed size pattern definitions.

The meaning of the various lines is:

Line 0001 gives the version number of the grammar this file follows. For variable size pattern files the version V3.2 is mandatory!

Line 0002 is information for the book keeping of the user only. Usually the time of the generation of the pattern file is given here. The string 'generated at' is mandatory !

Line 0004 gives the number of patterns defined in this file. The number of subpatterns is not specified, since it depends on the size of the network. Remember: The same pattern may be used by different sized networks resulting in varying numbers of subpatterns!

Line 0005 **CAUTION !** This variable does **NOT** give the number of input units but the size C of the fixed dimension. For TDNNs this would be the (invariant) number of features, for a picture it would be the number of values per pixel (i.e. a bitmap picture would have size 1, an RGB picture size 3).

Line 0006 corresponds to line 0005 for the output pattern

Line 0007 this line specifies the number of variable input dimensions I. With fixed size patterns 0 has to be specified.

Line 0008 this line specifies the size of the largest pattern in this pattern set. It is required for parsing and storage allocation purposes. The number of entries in the list has to match the number given in line 0007, if 0 was specified there an empty list (i.e. "[ ]")has to be given here.

Note: The lines 0007 and 0008 are pairwise mandatory, i.e. if one is given, the other has to be specified as well. Old pattern files do have neither one and can therefore still be read correctly.

Line 0009 corresponds to line 0007 for the output pattern. It specifies the number of variable output dimensions O.

Line 0010 corresponds to line 0008 for the output pattern.

Note: The lines 0009 and 0010 are again pairwise mandatory, i.e. if one is given, the other has to be specified as well. Old pattern files do have neither one and can therefore still be read correctly.

Line 0012 an arbitrary comment. All Text following the sign '#' in the same line is ignored.

Line 0013 this line has to be specified whenever I in line 0007 is $\neq 0$. It specifies the size of the following input pattern and is given as a list of integers separated by blanks and enclosed in [ ]. The values have to be given by descending dimensions row, i.e. [dimension_3 dimension_2 dimension_1] (here: [200 190]). Note that [200 190] is less than the maximum, which is specified in line 0008.

Line 0014 the first line of $\prod_{i=1}^{I} dimension_i * C$ activation values[2] (i.e. here 1*190 = 190 integer values). The values are expected to be stored as:
$dimension_{I-1}$ times
....
$dimension_1$ times
C values.

Line 0214 the last line of $\prod_{i=1}^{I} dimension_i * C$ activation values (i.e. here the 200^{th} line)

Line 0215 corresponds to line 0012 for the output pattern.

Line 0216 corresponds to line 0013 for the output pattern.

Line 0217 corresponds to line 0014 for the output pattern.

Once the patterns are loaded into the simulator, their handling can be controlled by using the control panel. For the handling of variable size patterns an additional subpattern panel is provided. The handling of patterns is described in conjunction with the control panel description in section 4.2.3. All these explanations are intended for fixed sized patterns, but also hold true for variable size patterns so they are not repeated here.

The additional functionality necessary for dealing with variable size patterns is provided by the subpat panel depicted in figure 5.1.

A subpattern is defined as the number of input and output activations that match the number of input and output units of the network. The size and shape of the subpattern must be defined in the subpat panel.

Note: A correct subpattern must be defined before any learning, propagation or recall function can be executed.

Note: For a network with 30 input units input subpatterns of size 1x30, 2x15, 3x10, 5x6, 6x5, 10x3, 2x15, and 30x1 would all be valid and would be propagated correctly if $C = 1$.

[2] C is the value read from line 0005

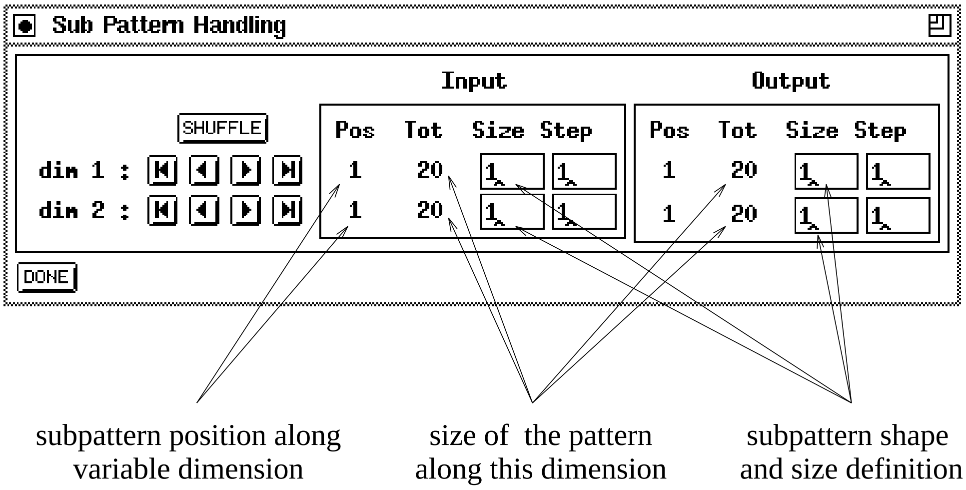

Figure 5.1: The handling panel for variable pattern sizes.

Is the position of the various input units important, however, (as in pictures) both size **and shape** have to match the network. Shape is not checked automatically, but has to be taken care of by the user! In the case of a color picture, where each pixel is represented by three values (RGB) C would be set to three and the set of possible combinations would shrink to 1x10, 2x5, 5x2, and 10x1.

Note: When loading a new pattern set, the list of activations is assigned to the units in order of ascending unit number. The user is responsible for the correct positioning of the units. When creating and deleting units their order is easily mixed up. This leads to unwanted graphical representations and the impression of the patterns being wrong. To avoid this behavior always make sure to have the lowest unit number in the upper left corner and the highest in the lower right. To avoid these problems use BIGNET for network creation.

Once a subpattern is defined, the user can scroll through the pattern along every dimension using the buttons , , , and . The step size used for scrolling when pressing the buttons and is determined by the input and output step fields for the various dimensions. The user can still as well browse through the pattern set using the arrow buttons of the control panel.

It is possible to load various pattern sets with a varying number of variable dimensions. The user is free to use any of them with the same network alternatively. When switching between these pattern sets the subpattern panel will automatically adapt to show the correct number of variable dimensions.

When stepping through the subpatterns, in learning, testing, or simply displaying, the resulting behavior is always governed by the input pattern. If the last possible subpattern within the current pattern is reached, the request for the next subpattern will automatically yield the first subpattern of the next pattern in both input and output layer. Therefore it is not possible to handle all subpatterns for training when there are not the same number of subpatterns in input and output layer available. By adjusting the step width accordingly, it should always be possible to achieve correct network behavior.

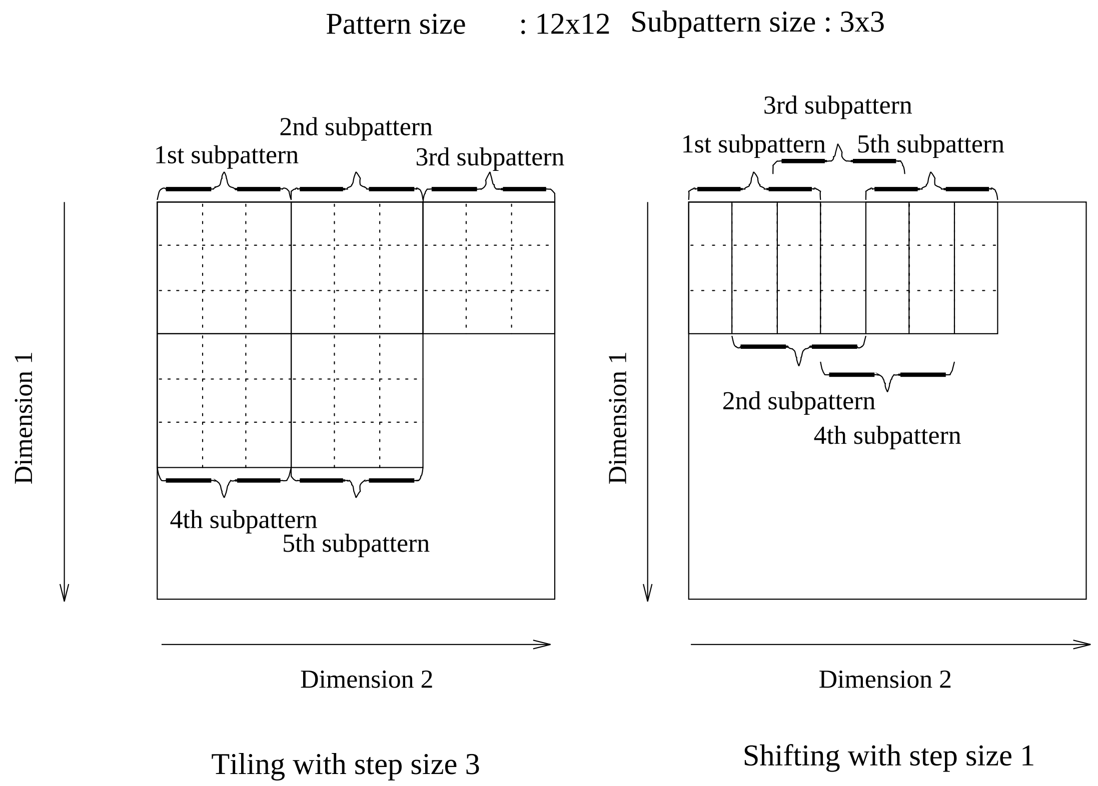

Figure 5.2: Tiling versus shifting of subpatterns.

The last possible subpattern in the above description is dependent from the settings of the subpattern panel. An example for a one-dimensional case would be: In a pattern of size 22 is the last subpattern with size = 3 and step = 5 the position 15. Changing the step width to 2 would lead to a last position of 18. In figure 5.2, the left pattern would have only 9 subpatterns, whereas the right one would have 49 !

The next reachable position with the current step width will always be a multiple of this step width. That is, if the step width is 4 and pattern position 8 is reached, a change of step width to 5 and a subsequent press of ▶ would result in position 10 (and not 13 as some might expect).

When selecting a step width, the user also has to remember whether the pattern should be divided in tiles or overlapping pieces. When implementing filter e.g., whether picture or others, a tiling stile will always be more appropriate, since different units are treated not concordantly.

It is the sole responsibility of the user to define the step width and the size of the subpattern correctly for both input and output. The user has to take care for the subpatterns to be correspondent. A wrong specification can lead to unpredictable learning behavior. The best way to check the settings is to press the TEST button, since exactly those subpatterns are thereby generated that will also be used for the training. By observing the reported position in the subpattern panel it can be verified whether meaningful values have been specified.

Chapter 6

Graphical Network Editor

The graphical user interface of SNNS has a network editor built in. With the network editor it is possible to generate a new network or to modify an existing network in various ways. There also exist commands to change the display style of the network.

As an introduction, operations on networks without sites will be discussed first, since they are easier to learn and understand. Operations that have a restricted or slightly different meaning for networks with sites are displayed with the extension *(Sites!)* in the following overview. These changes are discussed in detail in section 6.5.

As usual with most applications of X-Windows, the mouse must be in the window in which an input is to appear. This means that the mouse must be in the display window for editor operations to occur. If the mouse is moved in a display, the status indicator of the manager panel changes each time a new raster position in the display is reached.

Different displays of a network can be seen as different views of the same object. This means that all commands in one display may affect objects (units, links) in the other displays. Objects are moved or copied in a second display window in the same way as they are moved or copied in the first display window.

The editor operations are usually invoked by a sequence of 2 to 4 keys on the keyboard. They only take place when the last key of the command (e.g. deletion of units) is pressed. We found that for some of us, the fastest way to work with the editor was to move the mouse with one hand and to type on the keyboard with the other hand. Keyboard actions and mouse movement may occur at the same time, the mouse position is only relevant when the last key of the sequence is pressed.

The keys that are sufficient to invoke a part of a command are written in capital letters in the commands. The message line in the manager panel indicates the completed parts of the command sequence. Invalid keys are ignored by the editor.

As an example, if one presses the keys `U` for `Units` and `C` for `Copy` the status line changes as follows:

status line	**command**	**comment**
`>`	`Units`	operation on units

`Units>`	`Copy`	copying of units
`Units Copy>`		(the sequence is not completed yet)

To the left of the caret the fully expanded input sequence is displayed. At this place also a message is displayed when a command sequence is accepted and the corresponding operation is called. This serves as feedback, especially if the operation takes some time. If the operation completes quickly, only a short flicker of the text displayed can be seen. Some error messages appear in the confirmer, others in the message line.

6.1 Editor Modes

To work faster, three editor modes have been introduced which render the first key unnecessary. In normal mode all sequences are possible, in unit mode all sequences that deal with units (that start with `U`), and in link mode all command sequences that refer to links (i.e. start with `L`).

Example (continued from above):

status line	**command**	**comment**
`Units Copy>`	`Quit`	the input command may be cancelled any time
`>`	`Mode`	
`Mode>`	`Units`	enter unit mode
`Units>`	`Copy`	copying ...
`Units Copy>`	`Quit`	cancel again
`Units>`		`Quit` leaves the current mode unchanged
`Units>`	`Copy`	copying ...
`Units Copy>`	`Return`	return to normal mode
`>`		

The mode command is useful, if several unit or link commands are given in sequence. `Return` cancels a command, like `Quit` does, but also returns to normal mode.

6.2 Selection

6.2.1 Selection of Units

Units are selected by clicking on the unit with the left mouse button. On Black&White terminals, selected units are shown with crosses, on color terminals in a special, user defined, color. The default is yellow. By pressing and holding the mouse button down and moving the mouse, all units within a rectangular area can be selected, like in a number of popular drawing programs. It is not significant in what direction the rectangle is opened.

To remove a unit or group of units from a selection, one presses the `SHIFT` key on the keyboard while selecting the unit or group of units again. This undoes the previous selection for the specified unit or group of units. Alternatively, a single unit can be deselected with the right mouse button.

If the whole selection should be reset, one clicks in an empty raster position. The number of selected units is displayed at the bottom of the manager panel next to a stylized selection icon.

Example (setting activations of a group of units):

The activations of a group of units can be set to a specific value as follows: Enter the value in the activation value field of the target unit in the info panel. Select all units that should obtain the new value. Then enter the command to set the activation (`Units Set Activation`).

6.2.2 Selection of Links

Since it is often very hard to select a single link with the mouse in a dense web of links, in this simulator all selections of links are done with the reference to units. That is, links are selected via their source and target units. To select a link or a number of links, first a unit or a group of units must be selected in the usual way with the left mouse button (indicated by crosses through the units). Then the mouse pointer is moved to another unit. All links between the selected set of units and the unit under the mouse pointer during the last key stroke of the link command are then selected.

Example (deleting a group of links):

All links from one unit to several other units are deleted as follows: First select all target units, then point to the source unit with the mouse. Now the command `Links Delete from Source unit` deletes all the specified links.

As can be seen from the examples, for many operations three types of information are relevant: first a group of selected units, second the position of the mouse and the unit associated with this position and third some attributes of this unit which are displayed in the info panel. Therefore it is good practise to keep the info panel visible all the time.

In section 6.6 a longer example dialogue to build the well known XOR network (see also figure 3.1) is given which shows the main interaction principles.

6.3 Use of the Mouse

Besides the usual use of the mouse to control the elements of a graphical user interface (buttons, scroll bars etc.) the mouse is heavily used in the network editor. Many important functions like selection of units and links need the use of the mouse. The mouse buttons of the standard 3 button mouse are used in the following way within a graphic window:

- left mouse button:

Selects a unit. If the mouse is moved with the button pressed down, a group of units in a rectangular area is selected. If the `SHIFT` key is pressed at the same time, the units are deselected. The direction of movement with the mouse to open the rectangular area is not significant, i.e. one can open the rectangle from bottom right to top left, if convenient.

If the left mouse button is pressed together with the `CONTROL` key, a menu appears with all alternatives to complete the current command sequence. The menu items that display a trailing '!' indicate that the mouse position of the last command of a command sequence is important. The letter 'T' indicates that the target unit in the info panel plays a role. A (~) denotes that the command sequence is not yet completed.

- right mouse button:

  Undo of a selection. Clicking on a selected unit with the right mouse button only deselects this unit. Clicking on an empty raster position resets the whole selection.

- middle mouse button:

  Selects the source unit (on pressing the button down) and the target unit (on releasing the button) and displays them both in the info panel. If there is no connection between the two units, the target unit is displayed with its first source unit. If the button is pressed on a source unit and released over an empty target position, the link between the source and the current (last) target is displayed. If there is no such link the display remains unchanged. Conversely, if the button is pressed on an empty source position and released on an existing target unit, the link between the current (last) source unit and the selected target unit is displayed, if one exists. This is a convenient way to inspect links.

In order to indicate the position of the mouse even with a small raster size, there is always a sensitive area of at least 16x16 pixels wide.

6.4 Short Command Reference

The following section briefly describes the commands of the network editor. Capital letters denote the keys that must be hit to invoke the command in a command sequence.

The following commands are possible within any command sequence

- `Quit`: quit a command
- `Return`: quit a command and return to normal mode (see chapter 6.1)
- `Help`: get help information. A help window pops up (see chapter 4.2.10)

As already mentioned, some operations have a different meaning if there exist units with sites in a network. These operations are indicated with the suffix *(Sites!)* and are described in more detail in chapter 6.5. Commands that manipulate sites are also included in this overview. They start with the first command `Sites`.

- `Flags Safety`: sets/resets safety flag (a flag to prompt the user before units or links are deleted; additional question, if units with different subnet numbers are selected.)

1. Link Commands:

   - `Links Set`: sets all links between the selected units to the weight displayed in the info panel (independent of sites)
   - `Links Make` ...: creates or modifies connections
   - `Links Make Clique`: connects every selected unit with every other selected unit *(Sites!)*
   - `Links Make to Target unit`: creates links from all selected source units to a single target unit (under the mouse pointer) *(Sites!)*
   - `Links Make from Source unit`: creates links from a single source unit (under the mouse pointer) to all selected target units *(Sites!)*
   - `Links Make Double`: doubles all links between the selected units, i.e. generates two links (from source to target and from target to source) from each single link) *(Sites!)*
   - `Links Make Invers`: changes the direction of all links between the selected units *(Sites!)*
   - `Links Delete Clique`: deletes all links between all selected units *(Sites!)*
   - `Links Delete to Target unit`: deletes all incoming links from a selected group of units to a single target unit (under the mouse pointer) *(Sites!)*
   - `Links Delete from Source unit`: deletes all outgoing links from a single source unit (under the mouse pointer) to a selected group of units *(Sites!)*
   - `Links Copy Input`: copies all input links leading into the selected group of units as new input links to the target unit (under the mouse pointer) *(Sites!)*
   - `Links Copy Output`: copies all output links starting from the selected group of units as new output links of the source unit (under the mouse pointer) *(Sites!)*.
   - `Links Copy All`: copies all input and output links from the selected group of units as new input or output links to the unit under the mouse pointer *(Sites!)*
   - `Links Copy Environment`: copies all links between the selected units and the `TARGET` unit to the actual unit, if there exist units with the same relative distance *(Sites!)*

2. Site Commands:

   - `Sites Add`: add a site to all selected units
   - `Sites Delete`: delete a site from all selected units
   - `Sites Copy with No links`: copies the current site of the `Target` unit to all selected units. Links are not copied

- `Sites Copy with All links`: ditto, but with all links

3. Unit Commands:

- `Units Freeze`: freeze all selected units
- `Units Unfreeze`: reset freeze for all selected units
- `Units Set Name`: sets name to the name of `Target`
- `Units Set io-Type`: sets I/O type to the type of `Target`
- `Units Set Activation`: sets activation to the activation of `Target`
- `Units Set Initial activation`: sets initial activation to the initial activation of `Target`
- `Units Set Output`: sets output to the output of `Target`
- `Units Set Bias`: sets bias to the bias of `Target`
- `Units Set Function Activation`: sets activation function. Note: all selected units loose their default type (f-type)
- `Units Set Function Output`: sets output function Note: all selected units loose their default type (f-type)
- `Units Set Function Ftype`: sets default type (f-type)
- `Units Insert Default`: inserts a unit with default values. The unit has no links
- `Units Insert Target`: inserts a unit with the same values as the `Target` unit. The unit has no links
- `Units Insert Ftype`: inserts a unit of a certain default type (f-type) which is determined in a popup window
- `Units Delete`: deletes all selected units
- `Units Move`: all selected units are moved. The mouse determines the destination position of the `TARGET` unit (info-panel). The selected units and their position after the move are shown as outlines.
- `Units Copy ...`: copies all selected units to a new position. The mouse position determines the destination position of the `TARGET` unit (info-panel).
- `Units Copy All`: copies all selected units with *all* links
- `Units Copy Input`: copies all selected units with their input links
- `Units Copy Output`: copies all selected units and their output links
- `Units Copy None`: copies all selected units, but no links
- `Units Copy Structure ...`: copies all selected units and the link structure between these units, i.e. a whole subnet is copied

- `Units Copy Structure All`: copies all selected units, all links between them, and all input and output links to and from these units
- `Units Copy Structure Input`: copies all selected units, all links between them, and all input links to these units
- `Units Copy Structure Output`: copies all selected units, all links between them, and all output links from these units
- `Units Copy Structure None`: copies all selected units and all links between them
- `Units Copy Structure Back binding`: copies all selected units and all links between them and inserts additional links from the new to the corresponding original units *(Sites!)*
- `Units Copy Structure Forward binding`: copies all selected units and all links between them and inserts additional links from the original to the corresponding new units *(Sites!)*
- `Units Copy Structure Double binding`: ditto, but inserts additional links from the original to the new units and vice versa *(Sites!)*

4. Mode Commands:
   - `Mode Units`: unit mode, shortens command sequence if one wants to work with unit commands only. All subsequences after the `Units` command are valid then
   - `Mode Links`: analogous to `Mode Units`, but for link commands
5. Graphics Commands:
   - `Graphics All`: redraws the local window
   - `Graphics Complete`: redraws all windows
   - `Graphics Direction`: draws all links from and to a unit with arrows in the local window
   - `Graphics Links`: redraws all links in the local window
   - `Graphics Move`: moves the origin of the local window such that the `Target` unit is displayed at the position of the mouse pointer
   - `Graphics Origin`: moves the origin of the local window to the position indicated
   - `Graphics Grid`: displays a graphic grid at the raster positions in the local window
   - `Graphics Units`: redraws all units in the local window

6.5 Editor Commands

We now describe the editor commands in more detail. The description has the following form that is shown in two examples:

Links Make Clique (selection LINK : site-popup)

First comes the command sequence (**Links Make Clique**) which is invoked by pressing the keys L, M, and C in this order. The items in parentheses indicate that the command depends on the objects of a previous selection of a group of units with the mouse (selection), that it depends on the value of the LINK field in the info panel, and that a site-popup appears if there are sites defined in the network. The options are given in their temporal order, the colon ':' stands for the moment when the last character of the command sequence is pressed, i.e. the selection and the input of the value must precede the last key of the command sequence.

Units Set Activation (selection TARGET :)

The command sequence **Units Set Activation** is invoked by pressing the keys U, S, A, in that order. The items in parentheses indicate that the command depends on the selection of a group of units with the mouse (selection) which it depends on the value of the TARGET field and that these two things must be done before the last key of the command sequence is pressed.

The following table displays the meaning of the symbols in parenthesis:

selection	all selected units
:	now the last key of a command sequence is pressed
[unit]	the raster cursor is placed on a unit
[empty]	the raster cursor is placed on an empty position
default	the default values are used
TARGET	the TARGET unit field in the info panel must be set
LINK	the LINK field in the info panel must be set
site-links	only links to the current site in the info panel play a role
site	the current site in the info panel must be set
popup	a popup menu appears to ask for a value
site-popup	if there are sites defined in the network, a popup appears to choose the site for the operation
dest?	a raster position for a destination must be clicked with the mouse (e.g. in `Units Move`)

In the case of a `site-popup` a site for the operation can be chosen from this popup window. However, if one clicks the `DONE` button immediately afterwards, only the direct input without sites is chosen. In the following description, this direct input should be regarded as a special case of a site.

All newly generated units are assigned to all active layers in the display in which the command for their creation was issued.

The following keys are always possible within a command sequence:

- `Quit`: quit a command
- `Return`: quit and return to normal mode
- `Help`: get help information to the commands

A detailed description of the commands follows:

1. **Flags Safety** (:)

   If the `SAFETY`-Flag is set, then with every operation which deletes units, sites or links (`Units Delete` ... or `Links Delete` ...) a confirmer asks if the units, sites or links should really be deleted. If the flag is set, this is shown in the manager panel with a `safe` after the little flag icon. If the flag is not set, units, sites or links are deleted immediately. There is no undo operation for these deletions.

2. **Links Set** (selection LINK :)

   All link weights between the selected units are set to the value of the LINK field in the info panel.

3. **Links Make Clique** (selection LINK : site-popup)

   A full connection between all selected units is generated. Since links may be deleted selectively afterwards, this function is useful in many cases where many links in both directions are to be generated.

   If a site is selected, a complete connection is only possible if all units have a site with the same name.

4. **Links Make from Source unit** (selection [unit] : site-popup)

   Links Make to Target unit (selection [unit] : site-popup)

   Both operations connect all selected units with a single unit under the mouse pointer. In the first case, this unit is the source, in the second, it is the target. All links get the value of the LINK field in the info panel.

   If sites are used, only links to the selected site are generated.

5. **Links Make Double** (selection :)

   All unidirectional links become double (bidirectional) links. That is, new links in the opposite direction are generated. Immediately after creation the new links possess the same weights as the original links. However, the two links do not share the weight, i.e. subsequent training usually changes the similarity.

   Connections impinging on a site only become bidirectional, if the original source units has a site with the same name.

6. **Links Make Inverse** (selection :)

   All unidirectional links between all selected units change their direction. They keep their original value.

Connections leading to a site are only reversed, if the original source unit has a site of the same name. Otherwise they remain as they are.

7. **Links Delete Clique** (selection : site-popup)

   Links Delete from Source unit (selection [unit] : site-popup)

   Links Delete to Target unit (selection [unit] : site-popup)

   These three operations are the reverse of `Links Make` in that they delete the connections. If the safety flag is set (the word `safe` appears behind the flag symbol in the manager panel), a confirmer window forces the user to confirm the deletion.

8. **Links Copy Input** (selection [unit] :)

   Links Copy Output (selection [unit] :)

   Links Copy All (selection [unit] :)

   `Links Copy Input` copies all input links of the selected group of units to the single unit under the mouse pointer. If sites are used, incoming links are only copied if a site with the same name as in the original units exists.

   `Links Copy Output` copies all output links of the selected group of units to the single unit under the mouse pointer.

   `Links Copy All` Does both of the two operations above

9. **Links Copy Environment** (selection TARGET site-links [unit] :)

   This is a rather complex operation: `Links Copy Environment` tries to duplicate the links between all selected units and the current `TARGET` unit in the info panel at the place of the unit under the mouse pointer. The relative position of the selected units to the `TARGET` unit plays an important role: if a unit exists that has the same relative position to the unit under the mouse cursor as the `TARGET` unit has to one of the selected units, then a link between this unit and the unit under the mouse pointer is created.

   The result of this operation is a copy of the structure of links between the selected units and the TARGET unit at the place of the unit under the mouse pointer. That is, one obtains the same topological structure at the unit under the mouse pointer.

   This is shown in figure 6.1. In this figure the structure of the `TARGET` unit and the four `Env` units is copied to the unit `UnderMousePtr`. However, only two units are in the same relative position to the `UnderMousePtr` as the `Env` units are to the `Target` unit, namely `corrEnv3` corresponding to `Env3` and `corrEnv4` corresponding to `Env4`. So only those two links from the units `corrEnv3` to `UnderMousePtr` and from `corrEnv4` to `UnderMousePtr` are generated.

10. **Sites Add** (selection : Popup)

    A site which is chosen in a popup window is added to all selected units. The command has no effect for all units which already have a site of this name (because the names of all sites of a unit must be different)

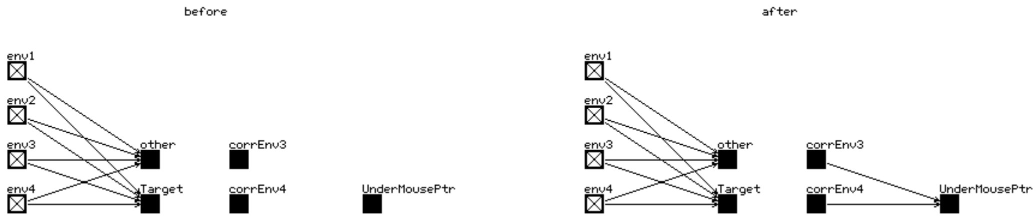

Figure 6.1: Example to `Links Copy Environment`

11. **Sites Delete** (selection : Popup)

    The site that is chosen in the popup window is deleted at all selected units that possess a site of this name. Also all links to this site are deleted. If the safety flag is set (in the manager panel the word `safe` is displayed behind the flag icon at the bottom), then a confirmer window forces the user to confirm the deletion first.

12. **Sites Copy with No links** (selection SITE :)

    Sites Copy with All links (selection SITE :)

    The current site of the `Target` unit is added to all selected units which do not have this site yet. Links are copied together with the site only with the command `Site Copy with All links`. If a unit already has a site of that name, only the links are copied.

13. **Units Freeze** (selection :)

    Units Unfreeze (selection :)

    These commands are used to freeze or unfreeze all selected units. Freezing means, that the unit does not get updated anymore, and therefore keeps its activation and output. Upon loading input units change only their activation, while keeping their output. For output units, this depends upon the setting of the pattern load mode. In the load mode `Output` only the output is set. Therefore, if frozen output units are to keep their output, another mode (`None` or `Activation`) has to be selected. A learning cycle, on the other hand, executes as if no units have been frozen.

14. **Units Set Name** (selection TARGET :)

    Units Set Initial activation (selection TARGET :)

    Units Set Output (selection TARGET :)

    Units Set Bias (selection TARGET :)

    Units Set io-Type (selection : Popup)

    Units Set Function Activation (selection : Popup)

    Units Set Function Output (selection : Popup)

Units Set Function F-type (selection : Popup)

Sets the specific attribute of all selected units to a common value. Types and functions are defined by a popup window. The operations can be aborted by immediately clicking the `DONE` button in the popup without selecting an element of the list.

The list item `special_X` for the command `Units Set io-Type` makes all selected units special while keeping their topologic type, .i.e.: a selected hidden unit becomes a special-hidden, a selected output becomes a special-output unit. The list item `non-special_X` performs the reverse procedure.

The remaining attributes are read from the corresponding fields of the `Target` unit in the info panel. The user can of course change the values there (without clicking the `SET` button) and then execute `Units Set ...`. A different approach would be to make a unit target unit (click on it with the middle mouse button) which already has the desired values. This procedure is very convenient, but works only if appropriate units already exist. A good idea might be to create a couple of such model units first, to be able to quickly set different attribute sets in the info panel.

15. **Units Insert Default** ([empty] default :)

    Units Insert Target ([empty] TARGET :)

    Units Insert F-type ([empty] : popup)

    This command is used to insert a unit with the IO-type `hidden`. It has no connections and its attributes are set according to the default values and the `Target` unit. With the command `Units Insert Default`, the unit gets no F-type and no sites. With `Units Insert F-type` an F-type and sites have to be selected in a popup window. `Units Insert Target` creates a copy of the target unit in the info panel. If sites/connections are to be copied as well, the command `Units Copy All` has to be used instead.

16. **Units Delete** (selection :)

    All selected units are deleted. If the safety flag is set (`safe` appears in the manager panel behind the flag symbol) the deletion has to be confirmed with the confirmer.

17. **Units Move** (selection TARGET : dest?)

    All selected units are moved. The `Target` unit is moved to the position at which the mouse button is clicked. It is therefore recommended to make one of the units to be moved target unit and position the mouse cursor over the target unit before beginning the move. Otherwise all moving units will have an offset from the cursor. This new position must not be occupied by an unselected unit, because a position conflict will result otherwise. All other units move in the same way relative to that position. The command is ignored, if:

    (a) the target position is occupied by an unselected unit, or

    (b) units would be moved to grid positions already taken by unselected units.

It might happen that units are moved beyond the right or lower border of the display. These units remain selected, as long as not all units are deselected (click the right mouse button to an empty grid position).

As long as no target is selected, the editor reacts only to `Return`, `Quit` or `Help`. Positioning is eased by displaying the unit outlines during the move. The user may also switch to another display. If this display has a different subnet number, the subnet number of the units changes accordingly. Depending upon layer and subnet parameters, it can happen that the moved units are not visible at the target.

If networks are generated externally, it might happen that several units lie on the same grid position. Upon selection of this position, only the unit with the smallest number is selected. With "Units Move" the user can thereby clarify the situation.

18. **Units Copy** ... (selection : dest?)

    Units Copy All

    Units Copy Input

    Units Copy Output

    Units Copy None

    This command is similar to `Units Move`. `Copy` creates copies of the selected units at the positions that would be assigned by `Move`. Another difference is that if units are moved to grid positions of selected units the command is ignored. The units created have the same attributes as their originals, but different numbers. Since unit types are copied as well the new units also inherit the activation function, output function and sites. There are four options regarding the copying of the links. If no links are copied, the new unit has no connections. If, for example, the input links are copied, the new units have the same predecessors as their originals.

19. **Units Copy Structure** ... (selection : dest?)

    Units Copy Structure All

    Units Copy Structure Input

    Units Copy Structure Output

    Units Copy Structure None

    Units Copy Structure ...binding (selection : dest? site-popup)

    Units Copy Structure Back binding

    Units Copy Structure Forward binding

    Units Copy Structure Double binding

    These commands are refinements of the general `Copy` command. Here, all links between the selected units are always copied as well. This means that the substructure is copied form the originals to the new units. On a copy without `Structure` these links would go unnoticed. There are also options, which additional links are to be copied. If only the substructure is to be copied, the command `Units Copy Structure None` is used.

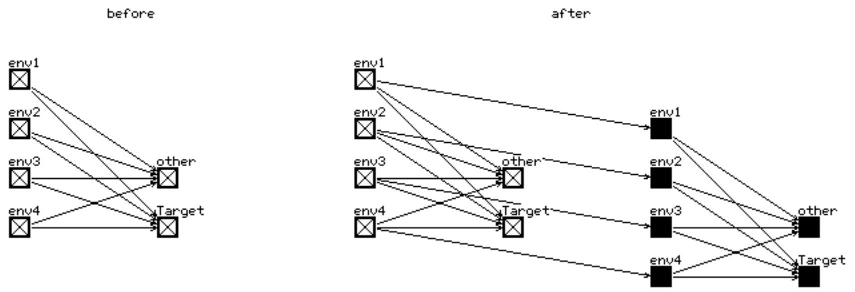

Figure 6.2: An Example for `Units Copy Structure with Forward binding`

The options with `binding` present a special feature. There, links between original and copied units are inserted automatically, in addition to the copied structure links. `Back`, `Forward` and `Double` specify thereby the direction of the links, where "back" means the direction towards the original unit. An example is shown in picture 6.2. If sites are used, the connections to the originals are assigned to the site selected in the popup. If not all originals have a site with that name, not all new units are linked to their predecessors.

With these various copy options, large, complicated nets with the same or similar substructures can be created very easily.

20. **Mode Units** (:)

    Mode Links (:)

    Switches to the mode **Units** or **Links**. All sequences of the normal modes are available. The keys `U` and `L` need not be pressed anymore. This shortens all sequences by one key.

21. **Units ...Return** (:)

    Links ...Return (:)

    Returns to normal mode after executing `Mode Units`.

22. **Graphics All** (:)

    Graphics Complete (:)

    Graphics Units (:)

    Graphics Links (:)

These commands initiate redrawing of the whole net, or parts of the net. With the exception of `Graphics Complete`, all commands affect only the current display. They are especially useful after links have been deleted.

23. **Graphic Direction** ([unit] :)

    This command assigns arrowheads to all links leading to/from the unit selected by the mouse. This is done independently from the setup values. XGUI, however, does not recall that links have been drawn. This means that, after moving a unit, these links remain in the window, if the display of links is switched off in the SETUP.

24. **Graphics Move** (TARGET [empty]/[unit] :)

    The origin of the window (upper left corner) is moved in a way that the `target` unit in the info panel becomes visible at the position specified by the mouse.

25. **Graphics Origin** ([empty]/[unit] :)

    The position specified by the mouse becomes new origin of the display (upper left corner).

26. **Graphics Grid** (:)

    This command draws a point at each grid position. The grid, however, is not refreshed, therefore one might have to redo the command from time to time.

6.6 Example Dialogue

A short example dialogue for the construction of an XOR network might clarify the use of the editor. First the four units are created. In the info panel the target name “input” and the `Target bias` “0” is entered.

Status Display	**Command**	**Remark**
`>`	`Mode Units`	switch on mode units
`Units>`		set mouse to position (3,5)
`Units>`	`Insert Target`	insert unit 1 with the attributes of the `Target` unit here. repeat for position (5,5).
`Units>`		name = “hidden”, bias = −2.88
`Units>`	`Insert Target`	position (3,3); insert unit 3
`Units>`		name = “output”, bias = −3.41
`Units>`	`Insert Target`	position (3,1); insert unit 4
`Units>`	`Return`	return to normal mode
`>`	`Mode Links`	switch on mode links
`Links>`		select both input units and set mouse to third unit

		("hidden")
`Links>`		specify weight "6.97"
`Links>`	`Make to Target`	create links
`Links>`		set mouse to unit 4 ("output"); specify weight "−5.24"
`Links>`	`Make to Target`	create links
`Links>`		deselect all units and select unit 3
`Links>`		set mouse to unit 4 and specify "11.71" as weight.
`Links>`	`Make to Target`	create links

Now the topology is defined. The only actions remaining are to set the IO types and the four patterns. To set the IO types, one can either use the command `Units Set Default io-type`, which sets the types according to the topological position of the units, or repeatedly use the command `Units Set io-Type`. The second option can be aborted by pressing the `Done` button in the popup window before making a selection.

Chapter 7

Network Creation Tools

SNNS provides five tools for easy creation of large, regular networks. All these tools carry the common name BigNet. They are called by clicking the button `BigNet` in the manager panel. This invokes the selection menu of figure 7.1, where the individual tools can be selected. This chapter gives a short indroduction to the handling of each of them.

Figure 7.1: The BigNet Menu Window

7.1 BigNet for Feed-Forward Networks

7.1.1 Terminology of the Tool BigNet

BigNet subdivides a net into several planes. The input layer, the output layer and every hidden layer are called a **plane** in the notation of BigNet. A plane is a two-dimensional array of units. Every single unit within a plane can be addressed by its coordinates. The unit in the upper left corner of every plane has the coordinates (1,1). A group of units within a plane, ordered in the shape of a square, is called a **cluster**. The position of a cluster is determined by the coordinates of its upper left corner and its expansion in the x direction (width) and y direction (height) (fig. 7.3).

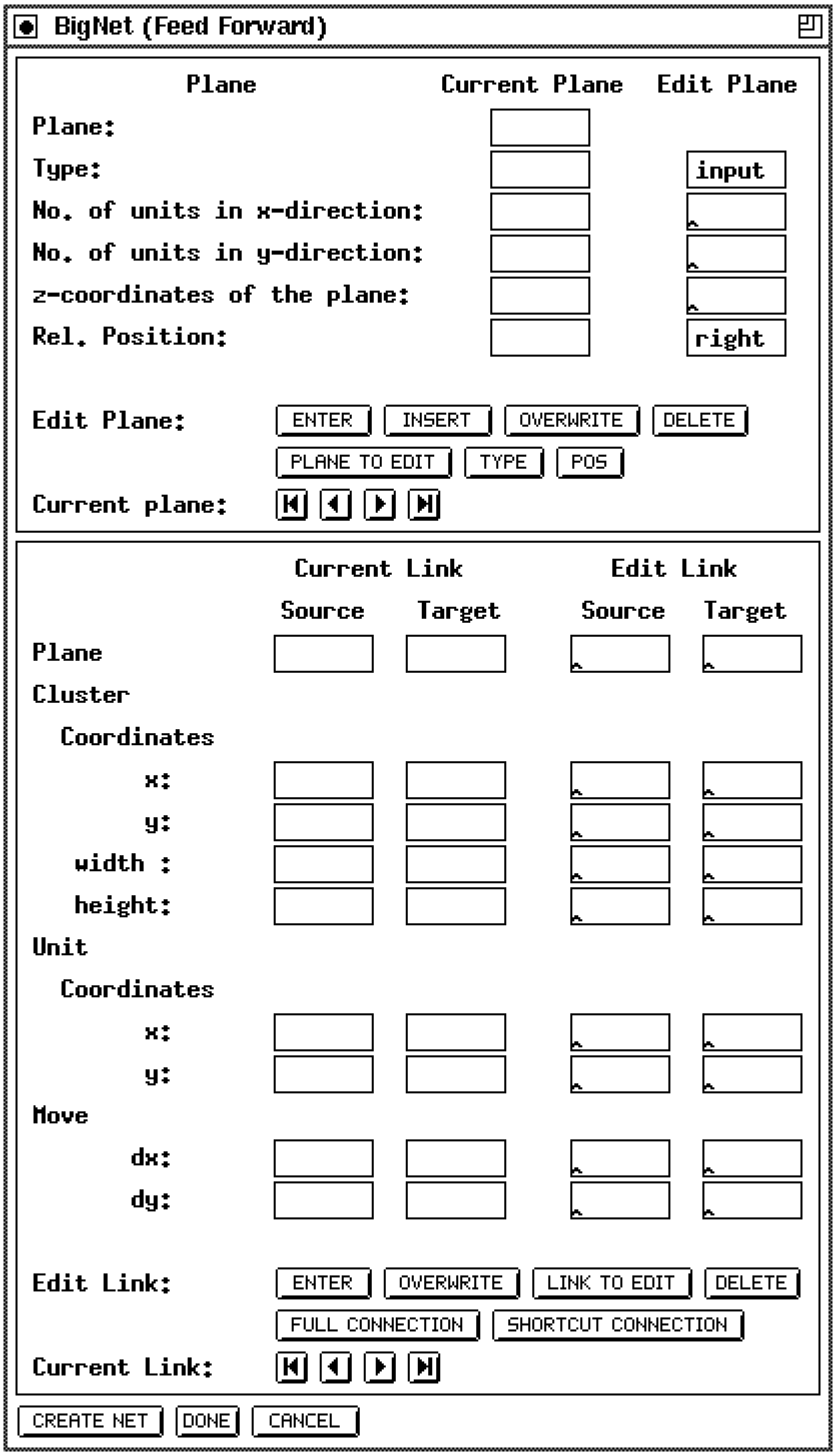

Figure 7.2: The BigNet window for Feed-Forward Networks

BigNet creates a net in two steps:

1. Edit net: This generates internal data structures in BigNet which describe the network but doesn't generate the network yet. This allows for easy modification of the network parameters before creation of the net.
   The net editor consists of two parts:
   (a) The plane editing part for editing planes. The input data is stored in the **plane list**.
   (b) The link editing part for editing links between planes. The input data is stored in the **link list**.
2. Generate net in SNNS: This generates the network from the internal data structures in BigNet.

Both editor parts are subdivided into an input part (Edit plane, Edit link) and into a display part for control purposes (Current plane, Current link). The input data of both editors is stored, as described above, in the plane list and in the link list. After pressing ENTER, INSERT, or OVERWRITE the input data is added to the corresponding editor list. In the control part one list element is always visible. The buttons , , , and enable moving around in the list. The operations DELETE, INSERT, OVERWRITE, CURRENT PLANE TO EDITOR and CURRENT LINK TO EDITOR refer to the current element. Input data is only entered in the editor list if it is correct, otherwise nothing happens.

7.1.2 Buttons of BigNet

`ENTER` : Input data is entered at the end of the plane or the link list.

`INSERT` : Input data is inserted in the plane list in front of the current plane.

`OVERWRITE` : The current element is replaced by the input data.

`DELETE` : The current element is deleted.

`PLANE TO EDIT` : The data of the current plane is written to the edit plane.

`LINK TO EDIT` : The data of the current link is written to the edit link.

`TYPE` : The type (input, hidden, output) of the units of a plane is determined.

`POS` : The position of a plane is always described relative (left, right, below) to the position of the previous plane. The upper left corner of the first plane is positioned at the coordinates (1,1) as described in Figure 7.4. BigNet then automatically generates the coordinates of the units.

`FULL CONNECTION` : A fully connected feed forward net is generated. If there are n planes numbered $1..n$ then every unit in plane i with $i > 0$ is connected with every unit in plane $i+1$ for all $1 \le i \le n-1$.

`SHORTCUT CONNECTION` : If there exist n planes $1 \ldots n$ then every unit in plane i with $1 \le i < n$ is connected with every unit in all planes j with $i < j \le n$.

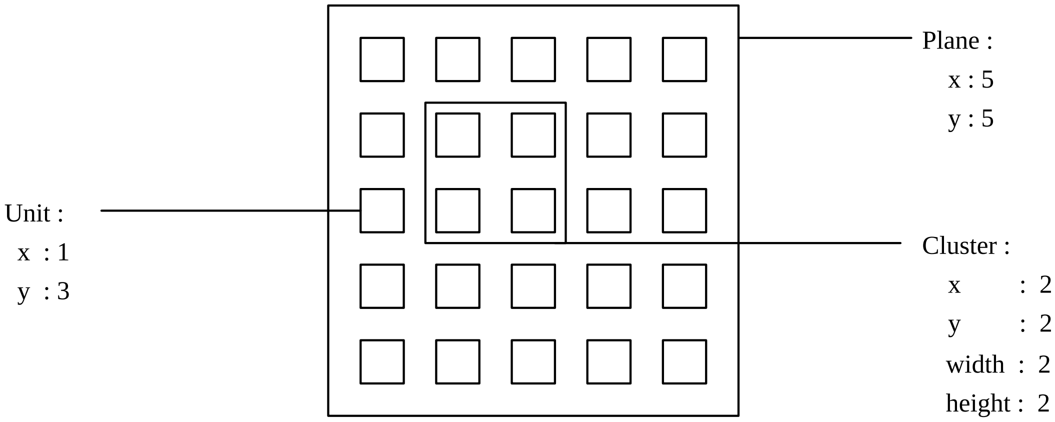

Figure 7.3: Clusters and units in BigNet

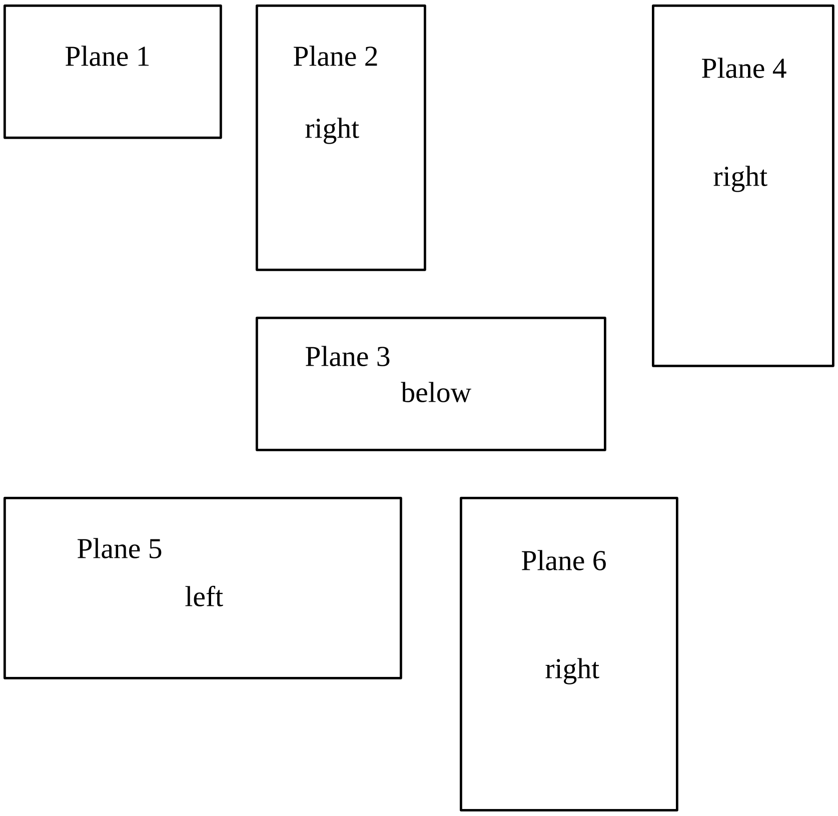

Figure 7.4: Positioning of the planes

`CREATE NET` : The net described by the two editors is generated by SNNS. The default name of the net is **SNNS_NET.net**. If a net with this name already exists a warning is issued before it is replaced.

`CANCEL` : All internal data of the editors is deleted.

`DONE` : Exit BigNet and return to the simulator windows.

7.1.3 Plane Editor

Every plane is characterized by the number of units in x and y direction. The unit type of a plane can be defined and changed by `TYPE`. The position of the planes is determined relative to the previous plane. The upper left corner of plane no. 1 is always positioned at the coordinates $(1, 1)$. Pressing `POS`, one can choose between 'left', 'right' and 'below'. Figure 7.4 shows the layout of a network with 6 planes which were positioned relative to their predecessors as indicated starting with plane 1.

Every plane is associated with a plane number. This number is introduced to address the planes in a clear way. The number is important for the link editor. The user cannot change this number.

In the current implementation the z coordinate is not used by BIGNET. It has been implemented for future use with the 3D visualization component.

7.1.4 Link Editor

A link always leads from a source to a target. To generate a fully connected net (connections from each layer to its succeeding layer, no shortcut connections), it is only sufficient to press the button `FULL CONNECTION` after the planes of the net are defined. Scrolling through the link list, one can see that every plane i is connected with the plane $i+1$. The plane number shown in the link editor is the same as the plane number given by the plane editor.

If one wants more complicated links between the planes one can edit them directly. There are nine different combinations to specify link connectivity patterns:

$$\text{Links from} \left\{ \begin{array}{c} \text{all units of a plane} \\ \text{all units of a cluster} \\ \text{a single unit} \end{array} \right\} \text{to} \left\{ \begin{array}{c} \text{all units of a plane} \\ \text{all units of a cluster} \\ \text{a single unit} \end{array} \right\}.$$

Figure 7.5 shows the display for the three possible input combinations with (all units of) a plane as source. The other combinations are similar. Note that both source plane and target plane must be specified in all cases, even if source or target consists of a cluster of units or a single unit. If the input data is inconsistent with the above rules it is rejected with a warning and not entered into the link list after pressing `ENTER` or `OVERWRITE`.

With the Move parameters one can declare how many steps a cluster or a unit will be moved in x or y direction within a plane after the cluster or the unit is connected with a target or a source. This facilitates the construction of receptive fields where all units of a cluster feed into a single target unit and this connectivity pattern is repeated in both directions with a displacement of one unit.

The parameter dx (delta-x) defines the step width in the x direction and dy (delta-y) defines the step width in the y direction. If there is no entry in dx or dy there is no movement in this direction. Movements within the source plane and the target plane is independent from each other. Since this feature is very powerful and versatile it will be illustrated with some examples.

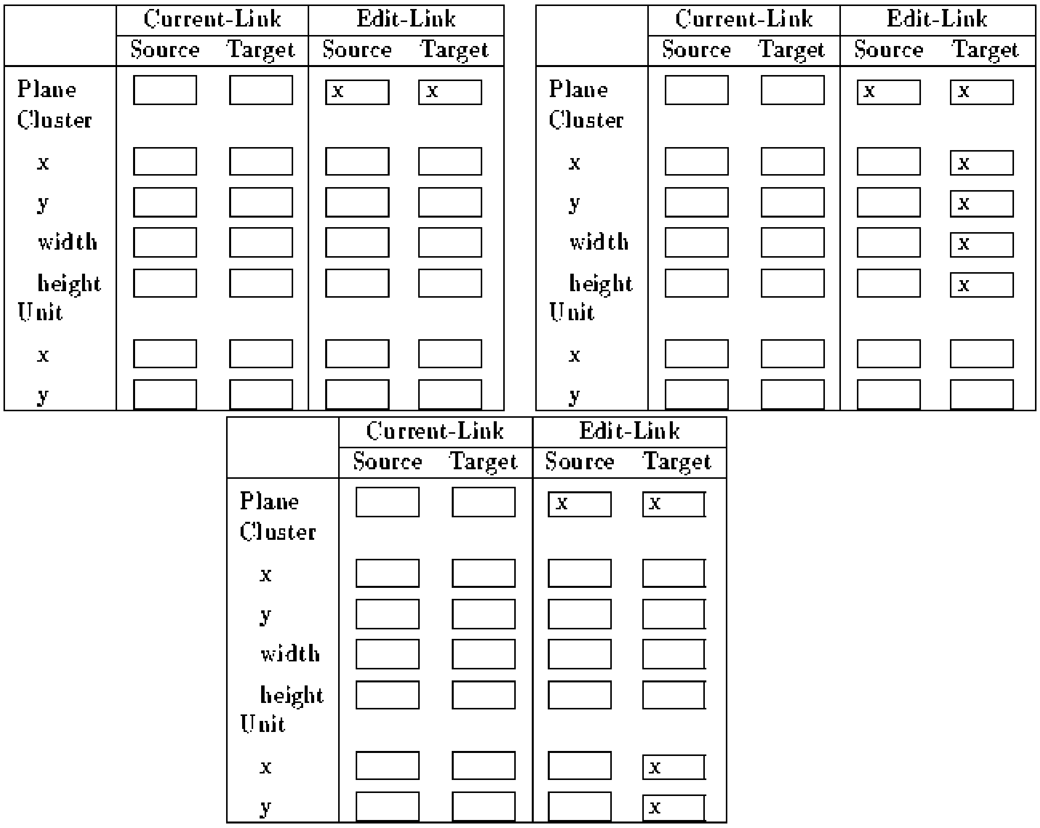

Figure 7.5: possible input combinations with (all units of) a plane as source, between 1) a plane and a plane, 2) a plane and a cluster, 3) a plane and a unit. Note that the target plane is specified in all three cases since it is necessary to indicate the target cluster or target unit.

Example 1: Receptive Fields in Two Dimensions

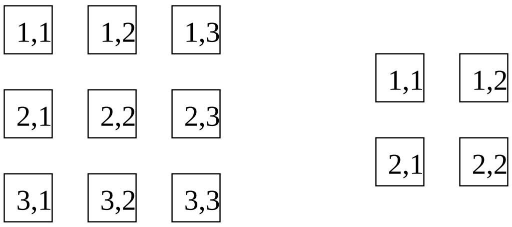

Figure 7.6: The net of example 1

There are two planes given (fig. 7.6). To realize the links

source: plane 1 (1,1), (1,2), (2,1) (2,2) $\longrightarrow$ target: plane 2 (1,1)
source: plane 1 (1,2), (1,3), (2,2) (2,3) $\longrightarrow$ target: plane 2 (1,2)
source: plane 1 (2,1), (2,2), (3,1) (3,2) $\longrightarrow$ target: plane 2 (2,1)
source: plane 1 (2,2), (2,3), (3,2) (3,3) $\longrightarrow$ target: plane 2 (2,2)

between the two planes, the move data shown in figure 7.7 must be inserted in the link editor.

	Current-Link		Edit-Link	
	Source	Target	Source	Target
Plane			1	2
Cluster				
x			1	
y			1	
width			2	
height			2	
Unit				
x				1
y				1
Move				
delta-x			1	1
delta-y			1	1

Figure 7.7: Example 1

First, the cluster (1,1), (1,2), (2,1) (2,2) is connected with the unit (1,1). After this step the source cluster and the target unit are moved right one step (this corresponds to dx = 1 for the source plane and the target plane). The new cluster is now connected with the new unit. The movement and connection building is repeated until either the source cluster or the target unit has reached the greatest possible x value. Then the internal unit pointer moves moves down one unit (this corresponds to dy = 1 for both planes) and back to the beginning of the planes. The "moving" continues in both directions until the boundaries of the two planes are reached.

Example 2: Moving in Different Dimensions

This time the net consists of three planes (fig. 7.9). To create the links

source: plane1 (1,1), (1,2), (1,3) $\longrightarrow$ target: plane 2 (1,1)
source: plane1 (2,1), (2,2), (2,3) $\longrightarrow$ target: plane 2 (1,2)
source: plane1 (3,1), (3,2), (3,3) $\longrightarrow$ target: plane 2 (1,3)
source: plane1 (1,1), (2,1), (3,1) $\longrightarrow$ target: plane 3 (1,1)
source: plane1 (1,2), (2,2), (3,2) $\longrightarrow$ target: plane 3 (1,2)
source: plane1 (1,3), (2,3), (3,3) $\longrightarrow$ target: plane 3 (1,3)

between the units one must insert the move data shown in figure 7.8. Every line of plane 1 is a cluster of width 3 and height 1 and is connected with a unit of plane 2, and every column of plane 1 is a cluster of width 1 and height 3 and is connected with a unit of plane 3. In this special case one can fill the empty input fields of "move" with any data because a movement in this directions is not possible and therefore these data is neglected.

	Current-Link		Edit-Link	
	Source	Target	Source	Target
Plane			1	2
Cluster				
x			1	
y			1	
width			3	
height			1	
Unit				
x				1
y				1
Move				
delta-x				1
delta-y			1	

	Current-Link		Edit-Link	
	Source	Target	Source	Target
Plane			1	3
Cluster				
x			1	
y			1	
width			1	
height			3	
Unit				
x				1
y				1
Move				
delta-x			1	1
delta-y				

Figure 7.8: Example 2

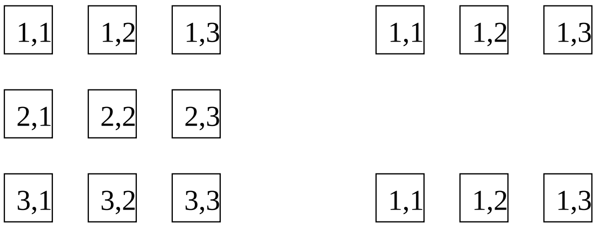

Figure 7.9: The net of example 2

7.1.5 Create Net

After one has described the net one must press `CREATE NET` to generate the net in SNNS. The weights of the links are set to the default value 0.5. Therefore one must initialize the net before one starts learning. The net created has the default name **SNNS_NET.net**. If a net already exists in SNNS a warning is issued before it is replaced. If the network generated happens to have two units with more than one connection in the same direction between them then SNNS sends the error message “Invalid Target”.

7.2 BigNet for Time-Delay Networks

The BigNet window for Time Delay networks (figure 7.10) consists of three parts: The Plane editor where the number, placement, and type of the units are defined, the link editor, where the connectivity between the layer is defined, and three control buttons at the bottom, to create the network, cancel editing, and close the window.

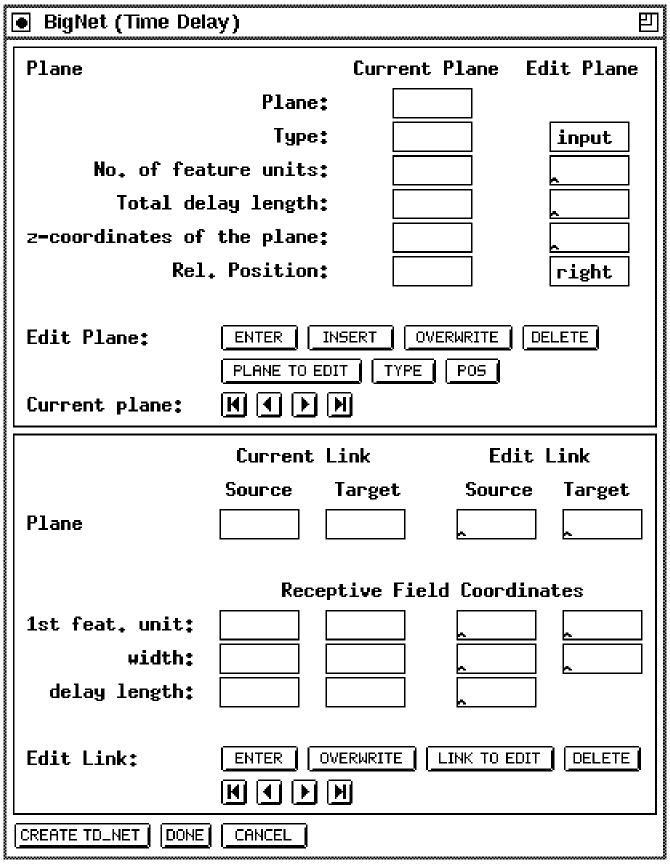

Figure 7.10: The BigNet window for Time Delay Networks

Since the buttons of this window carry mostly the same functionality as in the feed-forward case, refer to the previous section for a description of their use.

7.2.1 Terminology of Time-Delay BigNet

The following naming conventions have been adopted for the BigNet window. Their meaning may be clarified by figure 7.11.

- Receptive Field: The cluster of units in a layer totally connected to one row of units in the next layer.
- 1st feature unit: The starting row of the receptive field.

- width: The width of the receptive field.
- delay length: The number of significant delay steps of the receptive field. Must be the same value for all receptive fields in this layer.
- No. of feature units: The width of the current layer
- Total delay length: The length of the current layer. Total delay length times the number of feature units equals the number of units in this layer. Note that the total delay length must be the same as the delay length plus the total delay length of the next layer minus one!
- z-coordinates of the plane: gives the placing of the plane in space. This value may be omitted (default = 0).

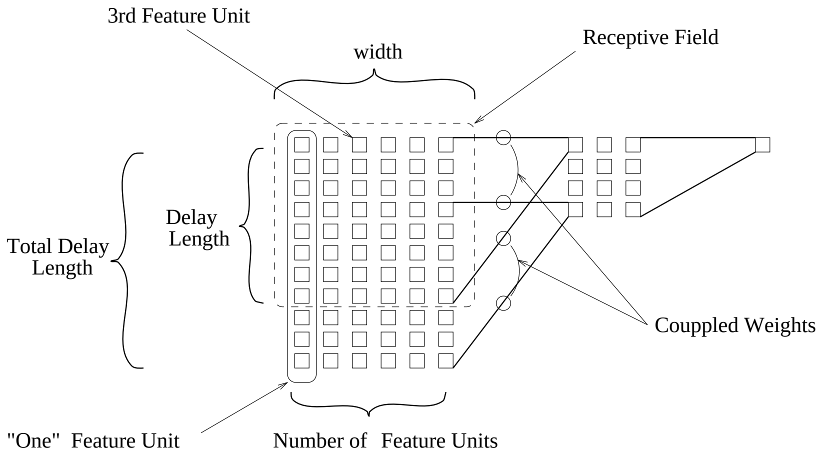

Figure 7.11: The naming conventions

7.2.2 Plane Editor

Just as in BigNet for feed-forward networks, the net is divided into several planes. The input layer, the output layer and every hidden layer are called a **plane** in the notation of BigNet. A plane is a two-dimensional array of units. Every single unit within a plane can be addressed by its coordinates. The unit in the upper left corner of every plane has the coordinates (1,1).

See 7.1.3 for a detailed description.

7.2.3 Link Editor

In the link panel the connections special to TDNNs can be defined. In TDNNs links always lead from the receptive field in a source plane to one or more units of a target plane. Note, that a receptive field has to be specified only once for each plane and is automatically applied to all possible delay steps in that plane. figure 7.12 gives an example of a receptive field specification and the network created thereby.

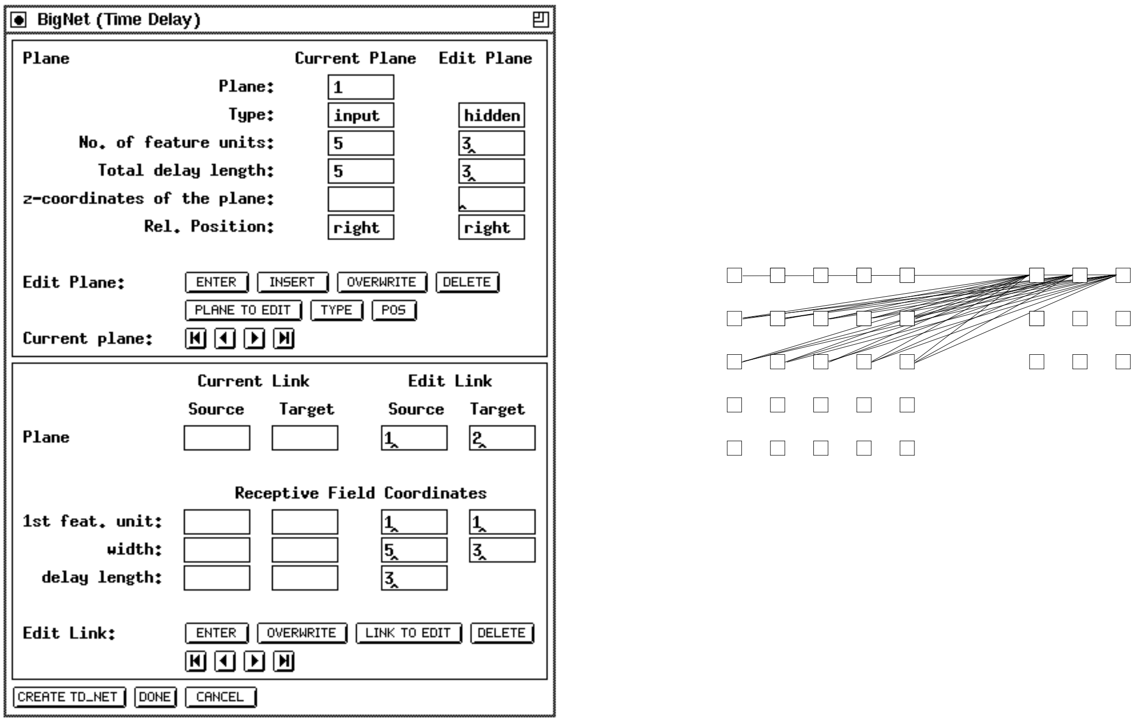

Figure 7.12: An example TDNN construction and the resulting network

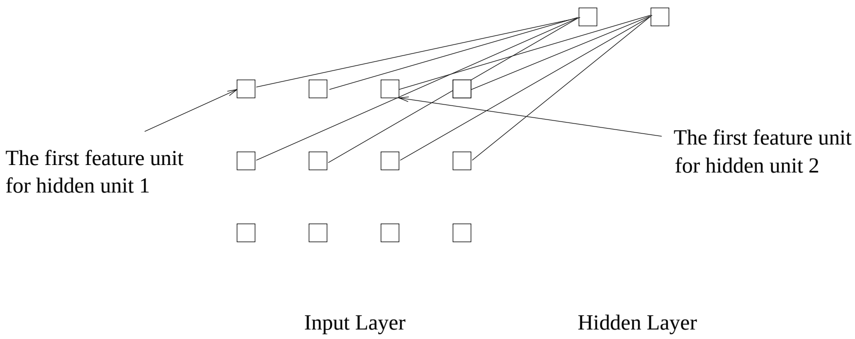

Figure 7.13: Two receptive fields in one layer

It is possible to specify seperate receptive fields for different feature units. With only one receptive field for all feature units, a "1" has to be specified in the input window for "1st feature unit:". For a second receptive field, the first feature unit should be the width of the first receptive field plus one. Of course, for all number of receptive fields, the sum of their width has to equal the number of feature units! An example network with two receptive fields is depicted in figure 7.13

7.3 BigNet for ART-Networks

The creation of the ART networks is based on just a few parameters. Although the network topology for these models is rather complex, only four parameters for ART1 and ART2, and eight parameters for ARTMAP, have to be specified.

If you have selected the `ART 1`, `ART 2` or the `ARTMAP` button in the BigNet menu, one of the windows shown in figure 7.14 appears on the screen.

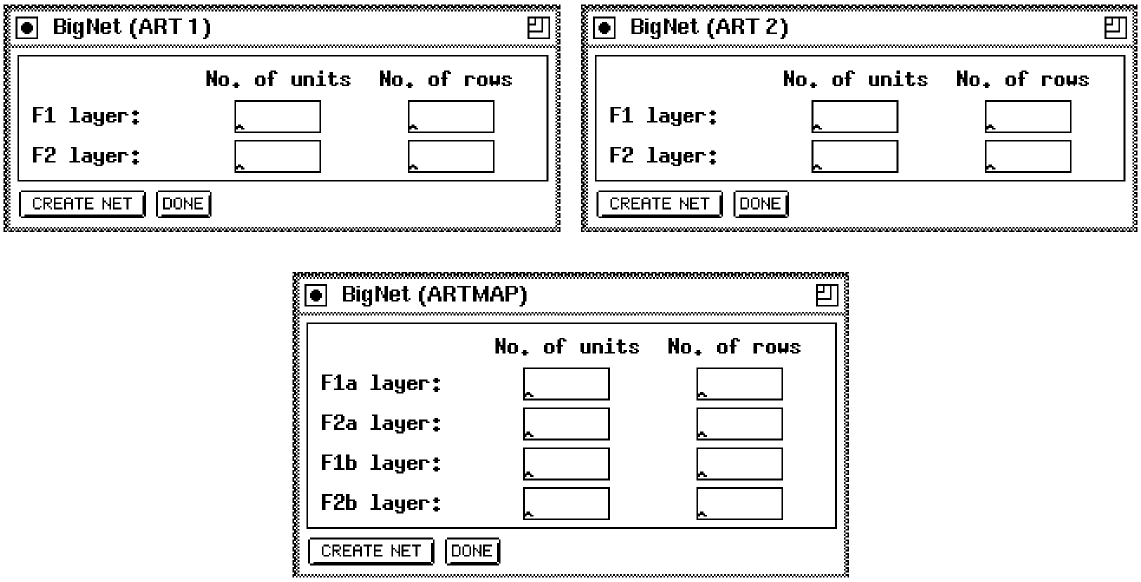

Figure 7.14: The BigNet windows for the ART models

The four parameters you have to specify for ART1 and ART2 are simple to choose. First you have to tell BigNet the number of units (N) the F_1 layer consists of. Since the F_0 layer has the same number of units, BigNet takes only the value for F_1.

Next the way how these N units to be displayed has to be specified. For this purpose enter the number of rows. An example for ART1 is shown in figure 7.15.

The same procedure is to be done for the F_2 layer. Again you have to specify the number of units M for the recognition part[1] of the F_2 layer and the number of rows.

Pressing the `CREATE NET` button will generate a network with the specified parameters. If a network exists when pressing `CREATE NET` you will be prompted to assure that you really want to destroy the current network. A message tells you if the generation terminated successfully. Finally press the `DONE` button to close the BigNet panel.

For ARTMAP things are slightly different. Since an ARTMAP network exists of two ART1 subnets (ART^a and ART^b), for both of them the parameters described above have to be specified. This is the reason, why BigNet (ARTMAP) takes eight instead of four parameters. For the MAP field the number of units and the number of rows is taken from the repective values for the F_2^b layer.

[1]The F_2 layer consists of three internal layers. See chapter 9.12.

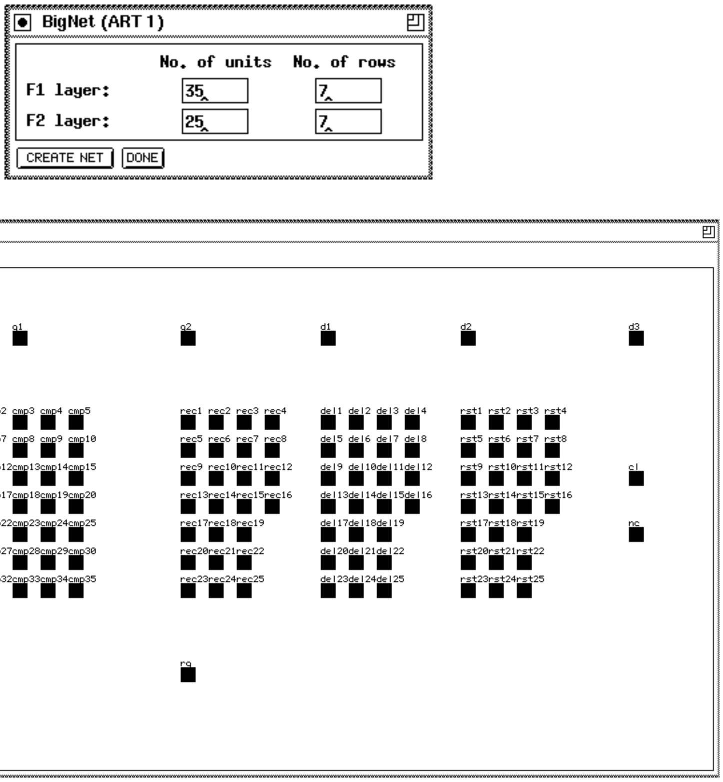

Figure 7.15: Example for the generation of an ART1 network. First the BigNet (ART1) panel is shown with the specified parameters. Next you see the created net as you can see it when using an SNNS display.

7.4 BigNet for Self-Organizing Maps

As described in chapter 9.13, it is recommended to create Kohonen-Self-Organizing Maps only by using either the **BigNet** network creation tool or **convert2snns** outside the graphical user interface. The SOM architecture consists of the component layer (input layer) and a two-dimensional map, called competitive layer. Since component layer and competitive layer are fully connected, each unit of the competitive layer represents a vector with the same dimension as the component layer.

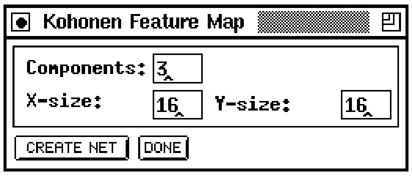

Figure 7.16: The BigNet window for the SOM architecture

To create a SOM, only 3 parameters have to be specified:

- *Components:* The dimension of each weight vector. It equals the number of input units.

- *X-size:* The width of the competitive layer. When learning is performed, the x-size value must be specified by the fifth learning parameter.

- *Y-size:* The length of the competitive layer. The number of hidden (competitive) units equals X-size ∗ Y-size.

If the parameters are correct (positive integers), pressing the CREATE NET button will create the specified network. If the creation of the network was successful a confirming message is issued. The parameters of the above example would create the network of figure 7.17. Eventually close the BigNet panel by pressing the DONE button.

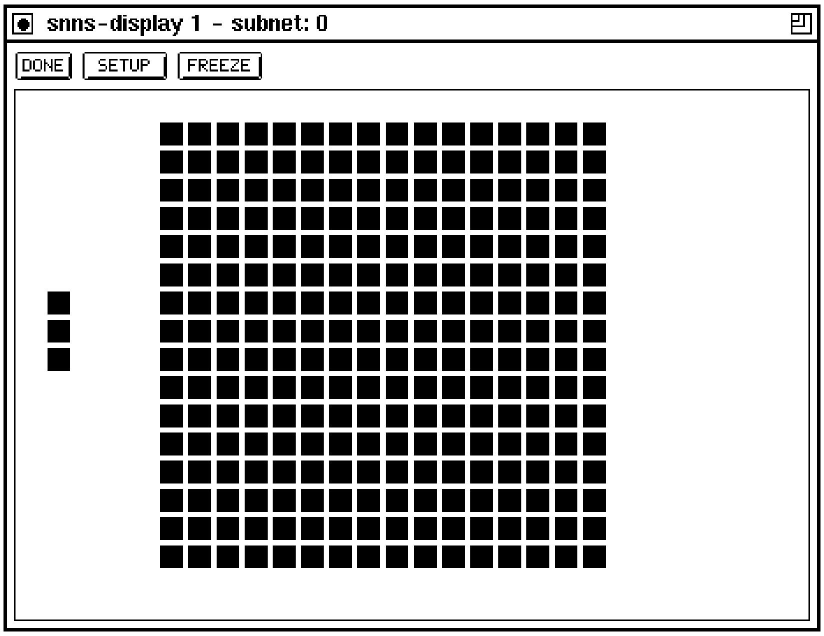

Figure 7.17: An Example SOM

7.5 BigNet for Autoassociative Memory Networks

The easiest way to create an autoassociative memory network is with the help of this bignet panel, although this type of network may also be constructed interactively with the graphical network editor. The architecture consists of the world layer (input layer) and a layer of hidden units, identical in size and shape to the world layer, called learning layer. Each unit of the world layer has a link to the corresponding unit in the learning layer. The learning layer is connected as a clique.

To create an autoassociative memory, only 2 parameters have to be specified:

- *X-size:* The width of the world and learning layers.

Figure 7.18: The BigNet Window for Autoassociative Memory Networks

- *Y-size:* The length of the world and learning layers. The number of units in the network equals 2 ∗ X-size ∗ Y-size.

If the parameters are correct (positive integers), pressing the CREATE NET button will create the specified network. If the creation of the network was successful a confirming message is issued. The parameters of the above example would create the network of figure 7.19. Eventually close the BigNet panel by pressing the DONE button.

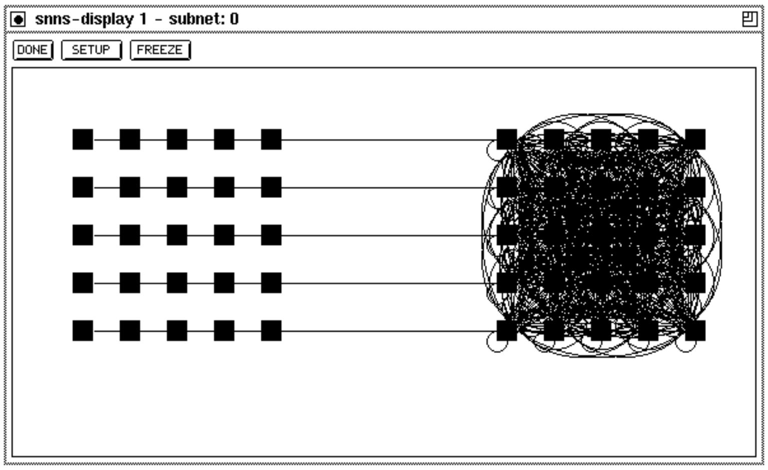

Figure 7.19: An Example Autoassociative Memory

7.6 BigNet for Partial Recurrent Networks

7.6.1 BigNet for Jordan Networks

The BigNet window for Jordan networks is shown in figure 7.20.

In the column `No. of Units` the number of units in the input, hidden and output layer have to be specified. The number of context units equals the number of output units. The units of a layer are displayed in several columns. The number of these columns is given by the value in the column `No. of Col.`. The network will be generated by pressing the `CREATE NET` button:

- The input layer is fully connected to the hidden layer,i.e. every input unit is connected to every unit of the hidden layer. The hidden layer is fully connected to the output layer.

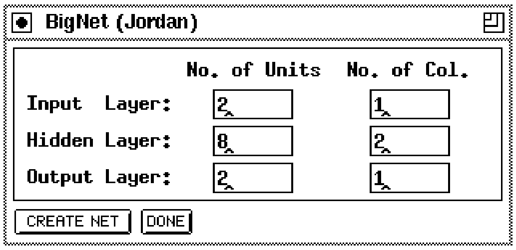

Figure 7.20: The BigNet Window for Jordan Networks

- Output units are connected to context units by recurrent 1-to-1-connections. Every context unit is connected to itself and to every hidden unit.
- Default activation function for input and context units is the identity function, for hidden and output units the logistic function.
- Default output function for all units is the identity function

To close the BigNet window for Jordan networks click on the DONE button.

7.6.2 BigNet for Elman Networks

By clicking on the ELMAN button in the BigNet menu, the BigNet window for Elman networks (see fig.7.21) is opened.

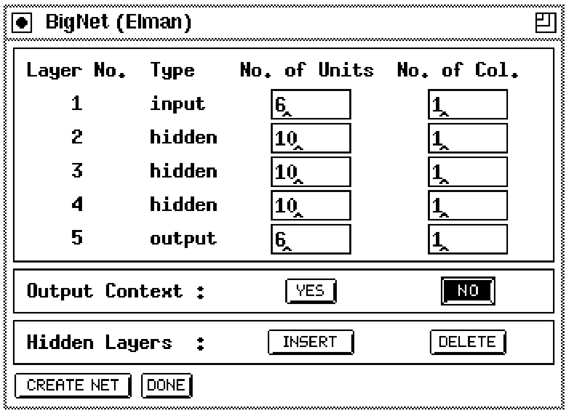

Figure 7.21: The BigNet Window for Elman Networks

The number of units of each layer has to be specified in the column `No. of Units`. Each hidden layer is assigned a context layer having the same size. The values in the column `No. of Col.` have the same meaning as the corresponding values in the BigNet window for Jordan networks.

The number of hidden layers can be changed with the buttons INSERT and DELETE. INSERT adds a new hidden layer just before the output layer. The hidden layer with the highest layer number can be deleted by pressing the DELETE button. The current implementation requires at least one and at most eight hidden layers. If the network is supposed to also contain a context layer for the output layer, the `YES` button has to be toggled, else the `NO` button. Press the `CREATE NET` button to create the net. The generated network has the following properties:

- The layer i is fully connected to the layer $i + 1$.
- Each context layer is fully connected to its hidden layer. A hidden layer is connected to its context layer with recurrent 1-to-1-connections.
- Each context unit is connected to itself.
- If there is a context layer assigned to the output layer, the same connection rules as for hidden layers are used.
- Default activation function for input and context units is the identity function, for hidden and output units the logistic function.
- Default output function for all units is the identity function

Click on the `DONE` button to close the BigNet window for Elman networks.

Chapter 8

Network Analyzing Tools

8.1 Inversion

Very often the user of a neural network asks what properties an input pattern must have in order to let the net generate a specific output. To help answer this question, the Inversion algorithm developed by J. Kindermann and A. Linden ([KL90]) was implemented in SNNS.

8.1.1 The Algorithm

The inversion of a neural net tries to find an input pattern that generates a specific output pattern with the existing connections. To find this input, the deviation of each output from the desired output is computed as error δ. This error value is used to approach the target input in input space step by step. Direction and length of this movement is computed by the inversion algorithm.

The most commonly used error value is the *Least Mean Square Error*. E^{LMS} is defined as

$$E^{LMS} = \sum_{p=1}^{n} [T_p - f(\sum_i w_{ij} o_{pi})]^2$$

The goal of the algorithm therefore has to be to minimize E^{LMS}.

Since the error signal δ_{pi} can be computed as

$$\delta_{pi} = o_{pi}(1 - o_{pi}) \sum_{k \in Succ(i)} \delta_{pk} w_{ik}$$

and for the adaption value of the unit activation follows

$$\triangle net_{pi} = \eta \delta_{pi} \quad \text{resp.} \quad net_{pi} = net_{pi} + \eta \delta_{pi}$$

In this implementation, a uniform pattern is applied to the input units in the first step, whose activation level depends upon the variable `input pattern`. This pattern is propagated through the net and generates the initial output $O^{(0)}$. The difference between this output vector and the target output vector is propagated backwards through the net as error signals $\delta_{i^{(0)}}$. This is analogous to the propagation of error signals in the backpropagation training, with the difference that no weights are adjusted here. When the error signals reach the input layer, they represent a gradient in input space, which gives the direction for the gradient descent. Thereby, the new input vector can be computed as

$$I^{(1)} = I^{(0)} + \eta * \delta_{i^{(0)}}$$

where η is the step size in input space, which is set by the variable `eta`.

This procedure is now repeated with the new input vector until the distance between the generated output vector and the desired output vector falls below the predefined limit of `delta_max`, when the algorithm is halted.

For a more detailed description of the algorithm and its implementation see [Mam92].

8.1.2 Inversion Display

The inversion algorithm is called by clicking the `INVERSION` button in the manager panel.

Picture 8.1 shows an example of the generated display.

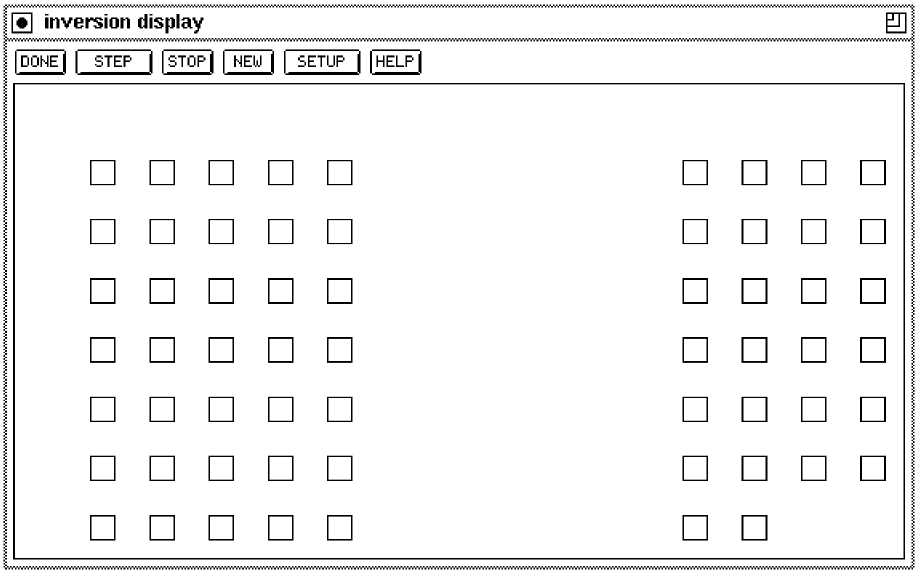

Figure 8.1: The Inversion Display

The display consists of two regions. The larger, lower part contains a sketch of the input and output units of the network, while the upper line holds a series of buttons. Their respective functions are:

1. `DONE`: Quits the inversion algorithm and closes the display.

2. `STEP`: Starts / Continues the algorithm. The program starts iterating by slowly changing the input pattern until either the STOP button is pressed, or the generated output pattern approximates the desired output pattern sufficiently well. Sufficiently well means that all output units have an activation which differs from the expected activation of that unit by at most a value of δ_{max}. This error limit can be set in the setup panel (see below). During the iteration run, the program prints status reports to stdout.

```
cycle 50 inversion error 0.499689 still 1 error unit(s)
cycle 100 inversion error 0.499682 still 1 error unit(s)
cycle 150 inversion error 0.499663 still 1 error unit(s)
cycle 200 inversion error 0.499592 still 1 error unit(s)
cycle 250 inversion error 0.499044 still 1 error unit(s)
cycle 269 inversion error 0.000000 0 error units left
```

   where cycle is the number of the current iteration, inversion error is the sum of the squared error of the output units for the current input pattern, and error units are all units that have an activation that differs more than the value of δ_{max} from the target activation.

3. `STOP`: Interrupts the iteration. The status of the network remains unchanged. The interrupt causes the current activations of the units to be displayed on the screen. A click to the STEP button continues the algorithm from its last state. Alternatively the algorithm can be reset before the restart by a click to the NEW button, or continued with other parameters after a change in the setup. Since there is no automatic recognition of infinite loops in the implementation, the STOP button is also necessary when the algorithm obviously does not converge.

4. `NEW` Resets the network to a defined initial status. All variables are assigned the values in the setup panel. The iteration counter is set to zero.

5. `SETUP`: Opens a pop-up window to set all variables associated with the inversion. These variables are:

`eta`	The step size for changing the activations. It should range from 1.0 to 10.0. Corresponds to the learning factor in backpropagation.
`delta_max`	The maximum activation deviation of an output unit. Units with higher deviation are called error units. A typical value of delta_max is 0.1.
`Input pattern`	Initial activation of all input units.
`2nd approx ratio`	Influence of the second approximation. Good values range from 0.2 to 0.8.

   A short description of all these variables can be found in an associated help window, which pops up on pressing HELP in the setup window.

   The variable `second approximation` can be understood as follows: Since the goal is to get a desired output, the first approximation is to get the network output as close as possible to the target output. There may be several input patterns generating the same output. To reduce the number of possible input patterns, the second approximation specifies a pattern the computed input pattern should approximate as well as possible. For a setting of 1.0 for the variable `Input pattern` the algorithm tries to keep as many input units as possible on a high activation, while a value of 0.0 increases the number of inactive input units. The variable `2nd approx ratio` defines then the importance of this input approximation.

It should be mentioned, however, that the algorithm is very unstable. One inversion run may converge, while another with only slightly changed variable settings may run indefinitely. The user therefore may have to try several combinations of variable values before a satisfying result is achieved. In general, the better the net was previously trained, the more likely is a positive inversion result.

6. `HELP`: Opens a window with a short help on handling the inversion display.

The network is displayed in the lower part of the window according to the settings of the last opened 2D–display window. Size, color, and orientation of the units are read from that display pointer.

8.1.3 Example Session

The inversion display may be called before or after the network has been trained. A pattern file for the network has to be loaded prior to calling the inversion. A target output of the network is defined by selecting one or more units in the 2D–display by clicking the middle mouse button. After setting the variables in the setup window, the inversion run is started by clicking the start button. At regular intervals, the inversion gives a status report on the shell window, where the progress of the algorithm can be observed. When there are no more error units, the program terminates and the calculated input pattern is displayed. If the algorithm does not converge, the run can be interrupted with the stop button and the variables may be changed. The calculated pattern can be tested for correctness by selecting all input units in the 2D–display and then deselecting them immediately again. This copies the activation of the units to the display. It can then be defined and tested with the usual buttons in the control panel. The user is advised to delete the generated pattern, since its use in subsequent learning cycles alters the behavior of the network which is generally not desirable.

Figure 8.2 shows an example of a generated input pattern (left). Here the minimum active units for recognition of the letter 'V' are given. The corresponding original pattern is shown on the right.

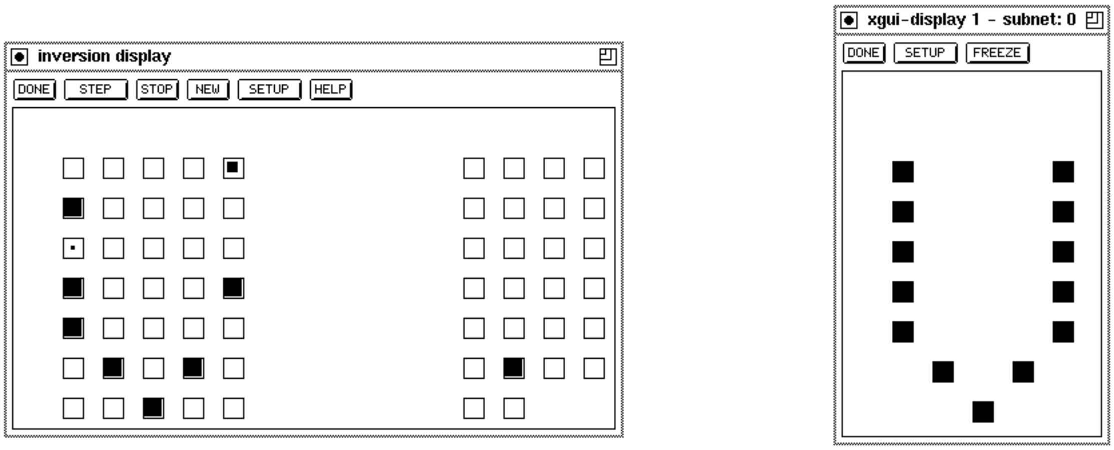

Figure 8.2: An Example of an Inversion Display (left) and the original pattern for the letter V

8.2 Network Analyzer

The network analyzer is a tool to visualize different types of graphs. An overview of these graphs is shown in table 8.1. This tool was especially developed for the prediction of time series with partial recurrent networks, but is also useful for regular feedforward networks. Its window is opened by selecting the entry `ANALYZER` in the menu under the `GUI` button.

The x-y graph is used to draw the activations or outputs of two units against each other. The t-y graph displays the activation (or output) of a unit during subsequent discrete time steps. The t-e graph makes it possible to visualize the error during subsequent discrete time steps.

Type	Axes
x–y	hor.: activation or output of a unit x ver.: activation or output of a unit y
t–y	hor.: time t ver.: activation or output of a unit y
t–e	hor.: time t ver.: error e

Table 8.1: The different types of graphs, which can be visualized with the network analyzer.

On the right side of the window, there are different buttons with the following functions:

`ON` : This button is used to "switch on" the network analyzer. If the Network Analyzer is switched on, every time a pattern has been propagated through the network, the network analyzer updates its display.

`LINE` : The points will be connected by a line, if this button is toggled.

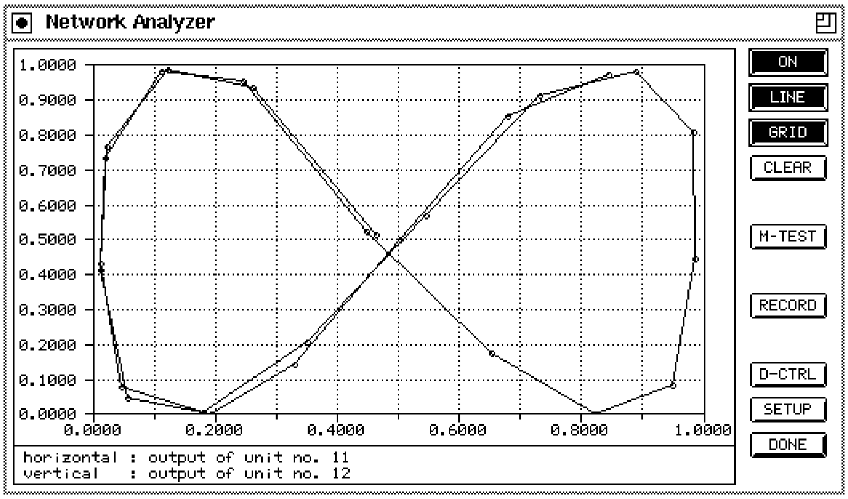

Figure 8.3: The Network Analyzer Window.

`GRID` : Displays a grid. The number of rows and columns of the grid can be specified in the network analyzer setup.

`CLEAR` : This button clears the graph in the display. The time counter will be reset to 1. If there is an active `M--TEST` operation, this operation will be killed.

`M-TEST` : A click on this button corresponds to several clicks on the `TEST` button in the control panel. The number n of `TEST` operations to be executed can be specified in the Network Analyzer setup. Once pressed, the button remains active until all n `TEST` operations have been executed or the `M--TEST` operation has been killed, e.g. by clicking the STOP button in the control panel.

`RECORD` : If this button activated, the points will not only be shown on the display, but their coordinates will also be saved in a file. The name of this file can be specified in the setup of the Network Analyzer.

`D-CTRL` : Opens the display control window of the Network Analyzer. The description of this window follows below.

`SETUP` : Opens the Network Analyzer setup window. The description of the setup follows in the next subsection.

`DONE` : Closes the network analyzer window. An active `M--TEST` operation will be killed.

8.2.1 The Network Analyzer Setup

The setup window can be opened by pressing the `SETUP` button on the right side of the network analyzer window. The shape of the setup window depends on the type of graph to display (see fig. 8.4).

The setup window consists of five parts (see fig. 8.4):

1. To select the type of graph (see table 8.1) to be displayed, press the corresponding button `X-Y` , `T-Y` or `T-E`.

2. The second part of the setup window is used to specify some attributes about the axes. The first line contains the values for the axes in horizontal direction, the second line these for the vertical axes. The columns `min` and `max` define the area to be displayed. The numbers of the units, whose activation or output values should be drawn have to be specified in the column `unit`. The last column `grid` the number of columns and rows of the grid can be varied. The labeling of the axes is dependent on these values, too.

3a The selection between showing the activation or the output of a unit along the x– or y–axes can be made here. To draw the output of a unit click on `OUT` and to draw the activation of a unit click on `ACT`.

3b Different types of error curves can be drawn:

$\sum_i \lvert t_i - o_i \rvert$	For each output unit the difference between the generated output and the teaching output is computed. The error is computed as the sum of the absolute values of the differences. If `AVE` is toggled, the result is divided by the number of output units, giving the average error per output unit.
$\sum_i \lvert t_i - o_i \rvert^2$	The error is computed as above, but the square of the differences is taken instead of the absolute values. With `AVE` the mean squared deviation is computed.
$\lvert t_j - o_j \rvert$	Here the deviation of only a single output unit is processed. The number of the unit is specified as `unit j`.

4. `m-test`: Specifies the number of `TEST` operations, which have to be executed when clicking on `M-TEST` button.

   `time`: Sets the time counter to the given value.

5. The name of the file, in which the visualized data can be saved by activating the `RECORD` button, can be specified here. The filename will be automatically extended by the suffix '.rec'. To change the filename, the `RECORD` button must not be activated.

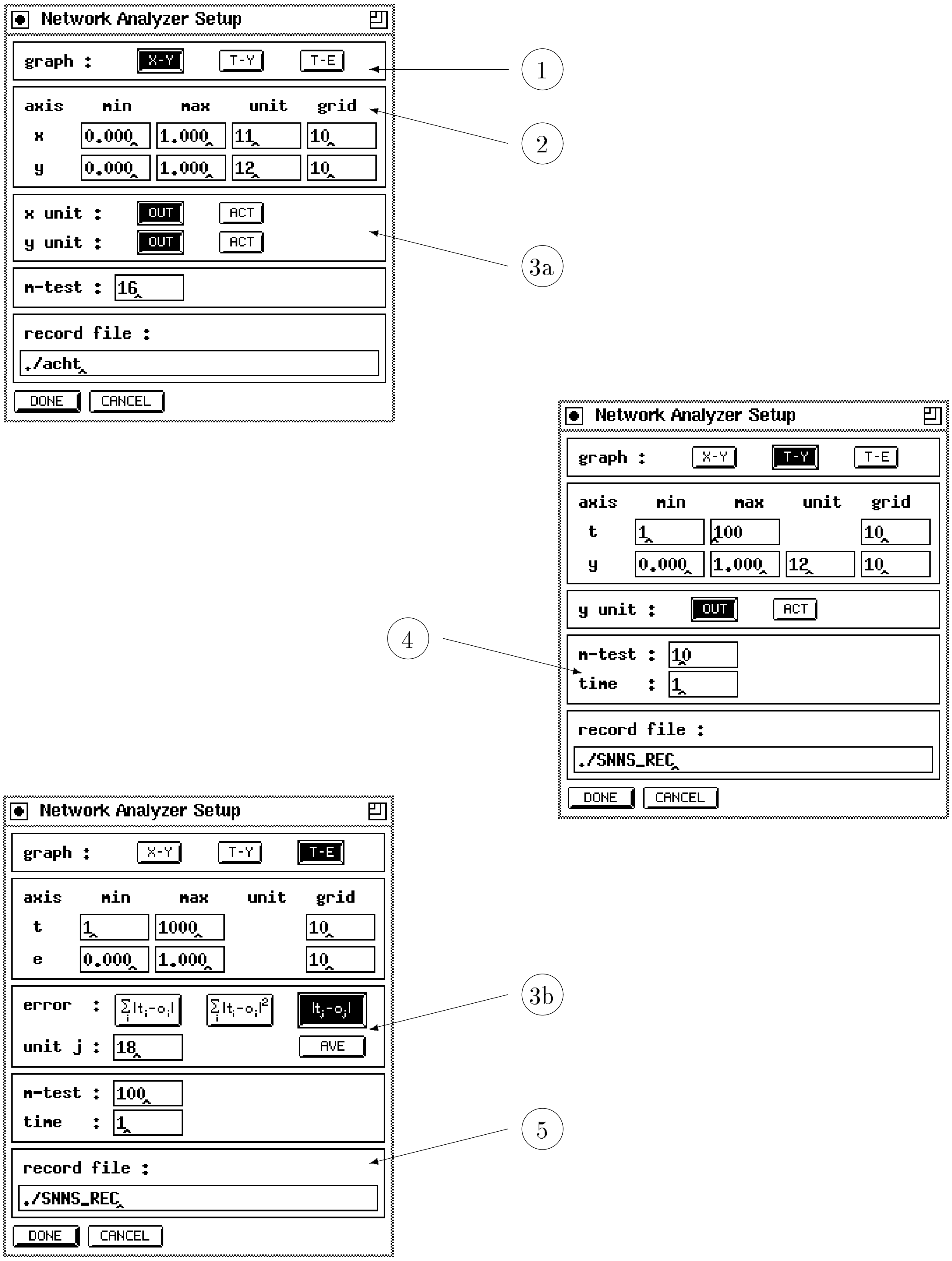

Figure 8.4: The Network Analyzer SetupWindows: the setup window for a x-y–graph (top), the setup window for a t-y–graph (middle) and the setup window for a t-e–graph (bottom).

When the setup is left by clicking on `CANCEL` all the changes made in the setup are lost.

When leaving the setup by pressing the DONE button, the changes will be accepted if no errors could be detected.

8.2.2 The Display Control Window of the Network Analyzer

The display control window appears, when clicking on D-CTRL button on the right side of the network analyzer window. This windows is used to easily change the area in the display of the network analyzer.

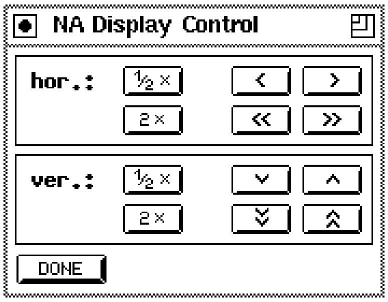

Figure 8.5: The display control window of the network analyzer

The buttons to change the range of the displayed area in horizontal direction have the following functions:

1/2 ×	:	The length of the interval is to be bisected. The lower bound remains.
2 ×	:	The length of the interval is to be doubled. The lower bound remains.
$<$	:	Shifts the range to the left by the width of a grid column.
$>$	:	Shifts the range to the right by the width of a grid column.
$\ll$	:	Shifts the range to the left by the length of the interval.
$\gg$	:	Shifts the range to the right by the length of the interval.

The buttons to change the range in vertical direction have a corresponding function. To close the display control window, press the DONE button.

Chapter 9

Neural Network Models and Functions

The following chapter introduces the models and learning functions implemented in SNNS. A strong emphasis is placed on the models that are less well known. They can not, however, be explained exhaustively here. We refer interested users to the literature.

9.1 Backpropagation Networks

9.1.1 Vanilla Backpropagation

The standard backpropagation learning algorithm introduced by [RM86] and described already in section 3.3 is implemented in SNNS. It is the most common learning algorithm. Its definition reads as follows:

$$\begin{aligned}
\Delta w_{ij} &= \eta\, \delta_j\, o_i \\
\delta_j &= \begin{cases} f_j'(net_j)(t_j - o_j) & \text{if unit } j \text{ is an output unit} \\ f_j'(net_j) \sum_k \delta_k w_{jk} & \text{if unit } j \text{ is a hidden unit} \end{cases}
\end{aligned}$$

This algorithm is also called *online backpropagation* because it updates the weights after every training pattern.

9.1.2 Enhanced Backpropagation

An enhanced version of backpropagation uses a momentum term and flat spot elimination. It is listed among the SNNS learning functions as `BackpropMomentum`.

The momentum term introduces the old weight change as a parameter for the computation of the new weight change. This avoids oscillation problems common with the regular backpropagation algorithm when the error surface has a very narrow minimum area. The new weight change is computed by

$$\Delta w_{ij}(t+1) = \eta * \delta_j * o_i + \alpha \Delta w_{ij}(t)$$

α is a constant specifying the influence of the momentum.

The effect of these enhancements is that flat spots of the error surface are traversed relatively rapidly with a few big steps, while the step size is decreased as the surface gets rougher. This adaption of the step size increases learning speed significantly.

Note that the old weight change is lost every time the parameters are modified, new patterns are loaded, or the network is modified.

9.1.3 Batch Backpropagation

Batch backpropagation has a similar formula as vanilla backpropagation. The difference lies in the time when the update of the links takes place. While in vanilla backpropagation an update step is performed after each single pattern, in batch backpropagation all weight changes are summed over a full presentation of all training patterns (one epoch). Only then, an update with the accumulated weight changes is performed. This update behavior is especially well suited for training pattern parallel implementations where communication costs are critical.

9.1.4 Backpropagation with Weight Decay

Weight Decay was introduced by P. Werbos ([Wer88]). It decreases the weights of the links while training them with backpropagation. In addition to each update of a weight by backpropagation, the weight is decreased by a part d of its old value. The resulting formula is

$$\Delta w_{ij}(t+1) = \eta\, \delta_j\, o_i - d\, w_{ij}(t)$$

The effect is similar to the pruning algorithms (see chapter 10). Weights are driven to zero unless reinforced by backpropagation. For further information, see [Sch94].

9.2 Quickprop

One method to speed up the learning is to use information about the curvature of the error surface. This requires the computation of the second order derivatives of the error function. Quickprop assumes the error surface to be locally quadratic and attempts to jump in one step from the current position directly into the minimum of the parabola.

Quickprop [Fah88] computes the derivatives in the direction of each weight. After computing the first gradient with regular backpropagation, a direct step to the error minimum is attempted by

$$\Delta(t+1)w_{ij} = \frac{S(t+1)}{S(t) - S(t+1)} \Delta(t) w_{ij}$$

where:

w_{ij}	weight between units i and j
$\Delta(t+1)$	actual weight change
$S(t+1)$	partial derivative of the error function by w_{ij}
$S(t)$	the last partial derivative

9.3 RPROP

9.3.1 Changes in Release 3.3

The implementation of Rprop has been changed in two ways: First, the implementation now follows a slightly modified adaptation scheme. Essentially, the backtracking step is no longer performed, if a jump over a minimum occurred. Second, a weight-decay term is introduced. The weight-decay parameter α (the third learning parameter) determines the relationship of two goals, namely to reduce the output error (the standard goal) and to reduce the size of the weights (to improve generalization). The composite error function is:

$$E = \sum (t_i - o_i)^2 + 10^{-\alpha} \sum w_{ij}^2$$

Important: Please note that the weight decay parameter α denotes the exponent, to allow comfortable input of very small weight-decay. A choice of the third learning parameter $\alpha = 4$ corresponds to a ratio of weight decay term to output error of $1 : 10000 (1 : 10^4)$.

9.3.2 General Description

Rprop stands for '**R**esilient back**prop**agation' and is a local adaptive learning scheme, performing supervised batch learning in multi-layer perceptrons. For a detailed discussion see also [Rie93], [RB93]. The basic principle of Rprop is to eliminate the harmful influence of the size of the partial derivative on the weight step. As a consequence, only the sign of the derivative is considered to indicate the *direction* of the weight update. The *size* of the weight change is exclusively determined by a weight-specific, so-called 'update-value' $\triangle_{ij}^{(t)}$:

$$\triangle w_{ij}^{(t)} = \begin{cases} -\triangle_{ij}^{(t)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ +\triangle_{ij}^{(t)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\ 0 & , \text{ else} \end{cases} \tag{9.1}$$

where $\frac{\partial E}{\partial w_{ij}}^{(t)}$ denotes the summed gradient information over all patterns of the pattern set ('batch learning').

It should be noted, that by replacing the $\triangle_{ij}(t)$ by a constant update-value $\triangle$, equation (9.1) yields the so-called 'Manhattan'-update rule.

The second step of Rprop learning is to determine the new update-values $\triangle_{ij}(t)$. This is based on a sign-dependent adaptation process.

$$\triangle_{ij}^{(t)} = \begin{cases} \eta^{+} * \triangle_{ij}^{(t-1)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ \eta^{-} * \triangle_{ij}^{(t-1)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\ \triangle_{ij}^{(t-1)} & , \text{ else} \end{cases} \tag{9.2}$$

$$\text{where } 0 < \eta^{-} < 1 < \eta^{+}$$

In words, the adaptation-rule works as follows: Every time the partial derivative of the corresponding weight w_{ij} changes its sign, which indicates that the last update was too big and the algorithm has jumped over a local minimum, the update-value $\triangle_{ij}^{(t)}$ is decreased by the factor η^{-}. If the derivative retains its sign, the update-value is slightly increased in order to accelerate convergence in shallow regions. Additionally, in case of a change in sign, there should be no adaptation in the succeeding learning step. In practice, this can be achieved by setting $\frac{\partial E}{\partial w_{ij}}^{(t-1)} := 0$ in the above adaptation rule (see also the description of the algorithm in the following section).

In order to reduce the number of freely adjustable parameters, often leading to a tedious search in parameter space, the increase and decrease factor are set to fixed values ($\eta^{-} := 0.5, \eta^{+} = 1.2$).

For Rprop tries to adapt its learning process to the topology of the error function, it follows the principle of 'batch learning' or 'learning by epoch'. That means, that weight-update and adaptation are performed after the gradient information of the whole pattern set is computed.

9.3.3 Parameters

The Rprop algorithm takes three parameters: the initial update-value $\triangle_0$, a limit for the maximum step size, $\triangle_{max}$, and the weight-decay exponent α (see above).

When learning starts, all update-values are set to an initial value $\triangle_0$. Since $\triangle_0$ directly determines the size of the first weight step, it should be chosen according to the initial values of the weights themselves, for example $\triangle_0 = 0.1$ (default setting). The choice of this value is rather uncritical, for it is adapted as learning proceeds.

In order to prevent the weights from becoming too large, the maximum weight-step determined by the size of the update-value, is limited. The upper bound is set by the second parameter of Rprop, $\triangle_{max}$. The default upper bound is set somewhat arbitrarily to $\triangle_{max} = 50.0$. Usually, convergence is rather insensitive to this parameter as well. Nevertheless, for some problems it can be advantageous to allow only very cautious (namely small) steps, in order to prevent the algorithm getting stuck too quickly in suboptimal local minima. The minimum step size is constantly fixed to $\triangle_{min} = 1e^{-6}$.

9.4 Backpercolation

Backpercolation 1 (Perc1) is a learning algorithm for feedforward networks. Here the weights are not changed according to the error of the output layer as in backpropagation, but according to a unit error that is computed separately for each unit. This effectively reduces the amount of training cycles needed.

The algorithm consists of five steps:

1. A pattern is propagated through the network and the global error is computed.
2. The gradient δ is computed and propagated back through the hidden layers as in backpropagation.
3. The error ϵ in the activation of each hidden neuron is computed. This error specifies the value by which the output of this neuron has to change in order to minimize the global error Err.
4. All weight parameters are changed according to ϵ.
5. If necessary, an adaptation of the error magnifying parameter λ is performed once every learning epoch.

The third step is divided into two phases: First each neuron receives a message $\Delta\mu$, specifying the proposed change in the activation of the neuron (message creation - MCR). Then each neuron combines the incoming messages to an optimal compromise, the internal error ϵ of the neuron (message optimization - MOP). The MCR phase is performed in forward direction (from input to output), the MOP phase backwards.

The internal error ϵ_k of the output units is defined as $\epsilon_k = \lambda(d_k - \phi_k)$, where λ is the global error magnification parameter.

Unlike backpropagation Perc1 does not have a learning parameter. Instead it has an error magnification parameter λ. This parameter may be adapted after each epoch, if the total mean error of the network falls below the threshold value θ.

When using backpercolation with a network in SNNS the initialization function `Random_-Weights_Perc` and the activation function `Act_TanH_Xdiv2` should be used.

9.5 Counterpropagation

9.5.1 Fundamentals

Counterpropagation was originally proposed as a pattern-lookup system that takes advantage of the parallel architecture of neural networks. Counterpropagation is useful in pattern mapping and pattern completion applications and can also serve as a sort of bidirectional associative memory.

When presented with a pattern, the network classifies that pattern by using a learned reference vector. The hidden units play a key role in this process, since the hidden layer performs a competitive classification to group the patterns. Counterpropagation works best on tightly clustered patterns in distinct groups.

Two types of layers are used: The hidden layer is a Kohonen layer with competitive units that do unsupervised learning; the output layer is a Grossberg layer, which is fully connected with the hidden layer and is not competitive.

When trained, the network works as follows. After presentation of a pattern in the input layer, the units in the hidden layer sum their inputs according to

$$\text{net}_j = \sum_i w_{ij} o_i$$

and then compete to respond to that input pattern. The unit with the highest net input wins and its activation is set to 1 while all others are set to 0. After the competition, the output layer does a weighted sum on the outputs of the hidden layer.

$$a_k = \text{net}_k = \sum_j w_{jk} o_j$$

Let c be the index of the winning hidden layer neuron. Since o_c is the only nonzero element in the sum, which in turn is equal to one, this can be reduced to

$$a_k = w_{ck}$$

Thus the winning hidden unit activates a pattern in the output layer.

During training, the weights are adapted as follows:

1. A winner of the competition is chosen in response to an input pattern.
2. The weights between the input layer and the winner are adjusted according to

   $$w_{ic}(t+1) = w_{ic}(t) + \alpha(o_i - w_{ic}(t))$$

   All the other weights remain unchanged.
3. The output of the network is computed and compared to the target pattern.
4. The weights between the winner and the output layer are updated according to

   $$w_{ck}(t+1) = w_{ck}(t) + \beta(o_k - w_{ck}(t))$$

   All the other weights remain unchanged.

9.5.2 Initializing Counterpropagation

For Counterpropagation networks three initialization functions are available: CPN_Rand_Pat, CPN_Weights_v3.2, and CPN_Weights_v3.3. See section 4.5 for a detailed description of these functions.

Note:
In SNNS versions 3.2 and 3.3 there was only the initialization function `CPN_Weights` available. Although it had the same name, there was a significant difference between the two.

The older version, still available now as `CPN_Weights_v3.2` selected its values from the hyper**cube** defined by the two initialization parameters. This resulted in an uneven distribution of these values after they had been normalized, thereby biasing the network towards a certain (unknown) direction. The newer version, still available now as `CPN_Weights_v3.3` selected its values from the hyper**sphere** defined by the two initialization parameters. This resulted in an even distribution of these values after they had been normalized. However it had the disadvantage of having an exponential time complexity, thereby making it useless for networks with more than about 15 input units. The influence of the parameters on these two functions is given below.

Two parameters are used which represent the minimum (a) and maximum (b) of the range out of which initial values for the second (Grossberg) layer are selected at random. The vector w_i of weights leading to unit i of the Kohonen layer are initialized as normalized vectors (length 1) drawn at random from part of a hyper-sphere (hyper-cube). Here, min and max determine which part of the hyper body is used according to table 9.1.

min (a)	max (b)	vectors out of ...
$a \geq 0$	$b \geq 0$	positive sector
$a \geq 0$	$b < 0$	whole hyper-sphere
$a < 0$	$b \geq 0$	whole hyper-sphere
$a < 0$	$b < 0$	negative sector

Table 9.1: Influence of minimum and maximum on the initialization of weight vectors for CPN and SOM.

9.5.3 Counterpropagation Implementation in SNNS

To use Counterpropagation in SNNS the following functions and variables have to be selected.

One of the above mentioned initialization functions, the update function `CPN_Order`, and the learning function `Counterpropagation`. The activation function of the units may be set to any of the sigmoidal functions available in SNNS.

9.6 Dynamic Learning Vector Quantization (DLVQ)

9.6.1 DLVQ Fundamentals

The idea of this algorithm is to find a natural grouping in a set of data ([SK92], [DH73]. Every data vector is associated with a point in a d-dimensional data space. The hope is that the vectors $\vec{x}$ of the same class form a cloud or a cluster in data space. The algorithm presupposes that the vectors $\vec{x}$ belonging to the same class w_i are distributed normally with a mean vector $\vec{\mu_i}$, and that all input vectors are normalized. To classify a feature vector $\vec{x}$ measure the Euclidian distance $||\vec{\mu} - \vec{x}||^2$ from $\vec{x}$ to all other mean vectors $\vec{\mu}$ and assign $\vec{x}$ to the class of the nearest mean. But what happens if a pattern $\vec{x_A}$ of class w_A is assigned to a wrong class w_B? Then for this wrong classified pattern the two mean vectors $\vec{\mu_A}$ and $\vec{\mu_B}$ are moved or trained in the following way:

- The reference vector $\vec{\mu_A}$ which the wrong classified pattern belongs to, and which is the nearest neighbor to this pattern, is moved a little bit towards this pattern.
- The mean vector $\vec{\mu_B}$, to which a pattern of class w_A is assigned wrongly, is moved away from it.

The vectors are moved using the rule:

$$w_{ij} = w_{ij} + \eta(o_i - w_{ij}).$$

where w_{ij} is the weight[1] between the output o_i of a input unit i and a output unit j. η is the learning parameter. By choosing it less or greater than zero, the direction of movement of a vector can be influenced.

The DLVQ algorithm works in the following way:

1. Load the (normalized) training data, and calculate for every class the mean vector μ. Initialize the net with these vectors. This means: Generate a unit for every class and initialize its weights with the corresponding values.
2. Now try to associate every pattern in the training set with a reference vector. If a trainings vector $\vec{x}$ of a class w_A is assigned to a class w_B then do the following:
   (a) Move the vector $\vec{\mu_A}$ which is nearest to $\vec{x_A}$ in its direction.
   (b) Move the mean vector $\vec{\mu_B}$, to which $\vec{x_A}$ is falsely assigned to away from it.

   Repeat this procedure until the number of correctly classified vectors no longer increases.
3. Now calculate, from the vectors of a class w_A associated with a wrong class w_B, a new prototype vector μ_A. For every class, choose one of the new mean vectors and add it to the net. Return to step 2.

9.6.2 DLVQ in SNNS

To start DLVQ, the learning function `DLVQ`, the update function `DLVQ_Update`, and the init function `DLVQ_Weights` have to be selected in the corresponding menus. The init functions of `DLVQ` differ a little from the normal function: if a DLVQ net is initialized, all hidden units are deleted.

As with learning rules CC and RCC the text field `CYCLE` in the control panel does **not** specify the number of learning cycles. This field is used to specify the maximal number of class units to be generated for each class during learning. The number of learning cycles is entered as the third parameter in the control panel (see below).

[1]Every mean vector $\vec{\mu}$ of a class is represented by a class unit. The elements of these vectors are stored in the weights between class unit and the input units.

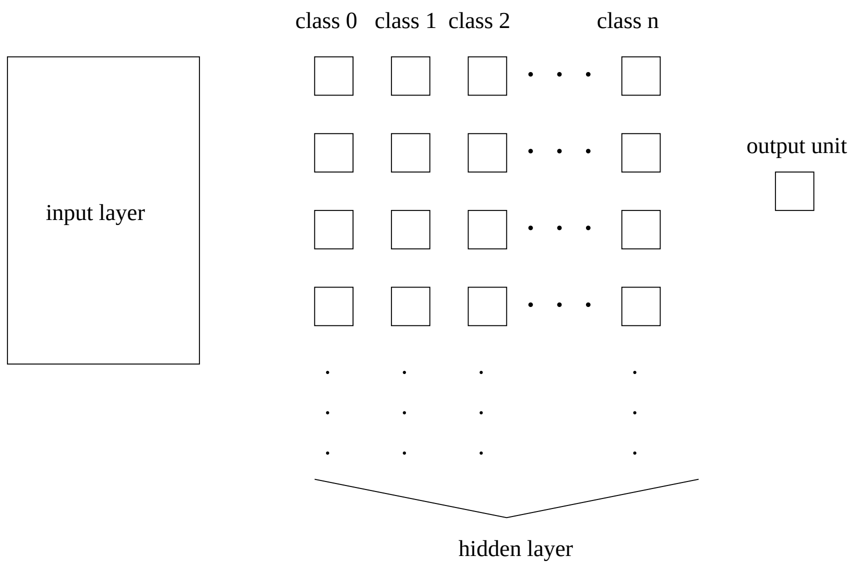

Figure 9.1: Topology of a net which was trained with DLVQ.

1. η^+: learning rate, specifies the step width of the mean vector $\vec{\mu_A}$, which is nearest to a pattern $\vec{x_A}$, towards this pattern. Remember that $\vec{\mu_A}$ is moved only, if $\vec{x_A}$ is not assigned to the correct class w_A. A typical value is 0.03.
2. η^-: learning rate, specifies the step width of a mean vector $\vec{\mu_B}$, to which a pattern of class w_A is falsely assigned to, away from this pattern. A typical value is 0.03. Best results can be achieved, if the condition $\eta^+ = \eta^-$ is satisfied.
3. Number of cycles you want to train the net before additive mean vectors are calculated.

If the topology of a net fits to the DLVQ architecture SNNS will order the units and layers from left to right independent in the following way: input layer, hidden layer output layer. The hidden layer itself is ordered by classes.

The output layer must consist of only one unit. At the start of the learning phase it does not matter whether the output layer and the input layer are connected. If hidden units exist, they are fully connected with the input layer. The links between these layers contain the values of the the mean vectors. The output layer and the hidden layer are fully connected. All these links have the value 1 assigned.

The output pattern contains information on which class the input pattern belongs to. The lowest class must have the name 0. If there are n classes, the n-th class has the name $n-1$. If these conditions are violated, an error occurs. Figure 9.1 shows the topology of a net. In the bias of every class unit its class name is stored. It can be retrieved by clicking on a class unit with right mouse button.

Note: In the first implementation of DLVQ the input patterns were automatically normalized by the algorithm. This step was eliminated, since is produced undesired behavior in some cases. Now the user has to take all necessary steps to normalize the input vectors correctly before loading them into SNNS.

9.6.3 Remarks

This algorithm was developed in the course of a masters thesis without knowledge of the original LVQ learning rules ([KKLT92]). Only later we found out that we had developed a new LVQ algorithm: It starts with the smallest possible number of hidden layers and adds new hidden units only when needed. Since the algorithm generates the hidden layer dynamically during the learning phase, it was called dynamic LVQ (DLVQ).

It is obvious that the algorithm works only if the patterns belonging to the same class have some similarities. Therefore the algorithm best fits classification problems such as recognition of patterns, digits, and so on. This algorithm succeeded in learning 10000 digits with a resolution of 16×16 pixels. Overall the algorithm generated 49 hidden units during learning.

9.7 Backpropagation Through Time (BPTT)

This is a learning algorithm for recurrent networks that are updated in discrete time steps (non-fixpoint networks). These networks may contain any number of feedback loops in their connectivity graph. The only restriction in this implementation is that there may be no connections between input units[2]. The gradients of the weights in the recurrent network are approximated using an feedforward network with a fixed number of layers. Each layer t contains all activations $a_i(t)$ of the recurrent network at time step t. The highest layer contains the most recent activations at time $t = 0$. These activations are calculated synchronously, using only the activations at $t = 1$ in the layer below. The weight matrices between successive layers are all identical. To calculate an exact gradient for an input pattern sequence of length T, the feedforward network needs $T + 1$ layers if an output pattern should be generated after the last pattern of the input sequence. This transformation of a recurrent network into a equivalent feedforward network was first described in [MP69], p. 145 and the application of backpropagation learning to these networks was introduced in [RHW86].

To avoid deep networks for long sequences, it is possible to use only a fixed number of layers to store the activations back in time. This method of *truncated backpropagation through time* is described in [Zip90] and is used here. An improved feature in this implementation is the combination with the quickprop algorithm by [Fah88] for weight adaption. The number of additional copies of network activations is controlled by the parameter `backstep`. Since the setting of `backstep` virtually generates a hierarchical network with `backstep` $+1$ layers and error information during backpropagation is diminished very rapidly in deep networks, the number of additional activation copies is limited by `backstep` ≤ 10.

There are three versions of backpropagation through time available:

BPTT: Backpropagation through time with online-update.
The gradient for each weight is summed over `backstep` copies between successive

[2]This case may be transformed into a network with an additional hidden unit for each input unit and a single connection with unity weight from each input unit to its corresponding hidden unit.

layers and the weights are adapted using the formula for backpropagation with momentum term after each pattern. The momentum term uses the weight change during the previous pattern. Using small learning rates `eta`, `BPTT` is especially useful to start adaption with a large number of patterns since the weights are updated much more frequently than in batch-update.

BBPTT: Batch backpropagation through time.
The gradient for each weight is calculated for each pattern as in `BPTT` and then averaged over the whole training set. The momentum term uses update information closer to the true gradient than in `BPTT`.

QPTT: Quickprop through time.
The gradient in quickprop through time is calculated as in `BBPTT`, but the weights are adapted using the substantially more efficient quickprop-update rule.

A recurrent network has to start processing a sequence of patterns with defined activations. All activities in the network may be set to zero by applying an input pattern containing only zero values. If such all-zero patterns are part of normal input patterns, an extra input unit has to be added for reset control. If this reset unit is set to 1, the network is in the free running mode. If the reset unit *and* all normal input units are set to 0, all activations in the network are set to 0 and all stored activations are cleared as well.

The processing of an input pattern $I(t)$ with a set of non-input activations $a_i(t)$ is performed as follows:

1. The input pattern $I(t)$ is copied to the input units to become a subset of the existing unit activations $a_i(t)$ of the whole net.
2. If $I(t)$ contains only zero activations, all activations $a_i(t+1)$ and all stored activations $a_i(t), a_i(t-1), \ldots, a_i(t-$ `backstep`$)$ are set to 0.0.
3. All activations $a_i(t+1)$ are calculated synchronously using the activation function and activation values $a_i(t)$.
4. During learning, an output pattern $O(t)$ is always compared with the output subset of the *new* activations $a_i(t+1)$.

Therefore there is exactly *one* synchronous update step between an input and an output pattern with the same pattern number.

If an input pattern has to be processed with more than one network update, there has to be a delay between corresponding input and output patterns. If an output pattern o^P is the n-th pattern after an input pattern i^P, the input pattern has been processed in $n+1$ update steps by the network. These $n+1$ steps may correspond to n hidden layers processing the pattern or a recurrent processing path through the network with $n+1$ steps. Because of this pipelined processing of a pattern sequence, the number of hidden layers that may develop during training in a fully recurrent network is influenced by the delay between corresponding input and output patterns. If the network has a defined hierarchical topology without shortcut connections between n different hidden layers, an output pattern should be the n-th pattern after its corresponding input pattern in the pattern file.

An example illustrating this relation is given with the delayed XOR network in the network file `xor-rec.net` and the pattern files `xor-rec1.pat` and `xor-rec2.pat`. With the patterns `xor-rec1.pat`, the task is to compute the XOR function of the previous input pattern. In `xor-rec2.pat`, there is a delay of 2 patterns for the result of the XOR of the input pattern. Using a fixed network topology with shortcut connections, the BPTT learning algorithm develops solutions with a different number of processing steps using the shortcut connections from the first hidden layer to the output layer to solve the task in `xor-rec1.pat`. To map the patterns in `xor-rec2.pat` the result is first calculated in the second hidden layer and copied from there to the output layer during the next update step[3].

The update function `BPTT-Order` performs the synchronous update of the network and detects reset patterns. If a network is tested using the TEST button in the control panel, the internal activations and the output activation of the output units are first overwritten with the values in the target pattern, depending on the setting of the button SHOW. To provide correct activations on feedback connections leading out of the output units in the following network update, all output activations are copied to the units initial activation values `i_act` after each network update and are copied back from `i_act` to `out` before each update. The non-input activation values may therefore be influenced before a network update by changing the initial activation values `i_act`.

If the network has to be reset by stepping over a reset pattern with the TEST button, keep in mind that after clicking TEST, the pattern number is increased first, the new input pattern is copied into the input layer second, and then the update function is called. So to reset the network, the current pattern must be set to the pattern directly preceding the reset pattern.

9.8 The Cascade Correlation Algorithms

Two cascade correlation algorithms have been implemented in SNNS, Cascade-Correlation and recurrent Cascade-Correlation. Both learning algorithms have been developed by Scott Fahlman ([FL91], [HF91], [Fah91]). Strictly speaking the cascade architecture represents a kind of meta algorithm, in which usual learning algorithms like Backprop, Quickprop or Rprop are embedded. Cascade-Correlation is characterized as a constructive learning rule. It starts with a minimal network, consisting only of an input and an output layer. Minimizing the overall error of a net, it adds step by step new hidden units to the hidden layer.

Cascade-Correlation is a supervised learning architecture which builds a near minimal multi-layer network topology. The two advantages of this architecture are that there is no need for a user to worry about the topology of the network, and that Cascade-Correlation learns much faster than the usual learning algorithms.

[3] If only an upper bound n for the number of processing steps is known, the input patterns may consist of windows containing the current input pattern together with a sequence of the previous $n-1$ input patterns. The network then develops a focus to the sequence element in the input window corresponding to the best number of processing steps.

9.8.1 Cascade-Correlation (CC)

9.8.1.1 The Algorithm

Cascade-Correlation (CC) combines two ideas: The first is the cascade architecture, in which hidden units are added only one at a time and do not change after they have been added. The second is the learning algorithm, which creates and installs the new hidden units. For each new hidden unit, the algorithm tries to maximize the magnitude of the correlation between the new unit's output and the residual error signal of the net.

The algorithm is realized in the following way:

1. CC starts with a minimal network consisting only of an input and an output layer. Both layers are fully connected.
2. Train all the connections ending at an output unit with a usual learning algorithm until the error of the net no longer decreases.
3. Generate the so-called candidate units. Every candidate unit is connected with all input units and with all existing hidden units. Between the pool of candidate units and the output units there are no weights.
4. Try to maximize the correlation between the activation of the candidate units and the residual error of the net by training all the links leading to a candidate unit. Learning takes place with an ordinary learning algorithm. The training is stopped when the correlation scores no longer improves.
5. Choose the candidate unit with the maximum correlation, freeze its incoming weights and add it to the net. To change the candidate unit into a hidden unit, generate links between the selected unit and all the output units. Since the weights leading to the new hidden unit are frozen, a new permanent feature detector is obtained. Loop back to step 2.

This algorithm is repeated until the overall error of the net falls below a given value. Figure 9.2 shows a net after 3 hidden units have been added.

9.8.1.2 Mathematical Background

The training of the output units tries to minimize the sum-squared error E:

$$E = \sum_p \frac{1}{2} \sum_o (y_{po} - t_{po})^2$$

where t_{po} is the desired and y_{po} is the observed output of the output unit o for a pattern p. The error E is minimized by gradient decent using

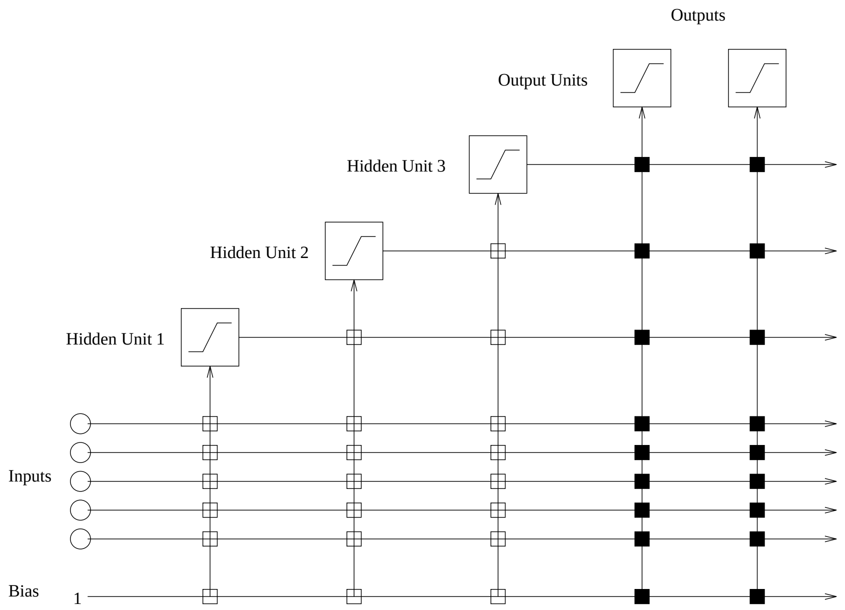

Figure 9.2: A neural net trained with cascade-correlation after 3 hidden units have been added. The vertical lines add all incoming activations. Connections with white boxes are frozen. The black connections are trained repeatedly.

$$
\begin{aligned}
e_{po} &= (y_{po} - t_{po}) f'_p(net_o) \\
\frac{\partial E}{\partial w_{io}} &= \sum_p e_{po} I_{ip},
\end{aligned}
$$

where f'_p is the derivative of an activation function of a output unit o and I_{ip} is the value of an input unit or a hidden unit i for a pattern p. w_{io} denominates the connection between an input or hidden unit i and an output unit o.

After the training phase the candidate units are adapted, so that the correlation C between the value y_{po} of a candidate unit and the residual error e_{po} of an output unit becomes maximal. The correlation is given by Fahlman with:

$$
\begin{aligned}
C &= \sum_o \left| \sum_p (y_{po} - \bar{y_o})(e_{po} - \bar{e_o}) \right| \\
&= \sum_o \left| \sum_p y_{po} e_{po} - \bar{e_o} \sum_p y_{po} \right| \\
&= \sum_o \left| \sum_p y_{po} (e_{po} - \bar{e_o}) \right|,
\end{aligned}
$$

where $\bar{y_o}$ is the average activation of a candidate unit and $\bar{e_o}$ is the average error of an output unit over all patterns p. The maximization of C proceeds by gradient ascent using

$$\delta_p = \sum_o \sigma_o (e_{po} - \bar{e_j}) f'_p$$
$$\frac{\partial C}{\partial w_i} = \sum_p \delta_p I_{pi},$$

where σ_o is the sign of the correlation between the candidate unit's output and the residual error at output o.

9.8.2 Pruned-Cascade-Correlation (PCC)

9.8.2.1 The Algorithm

The aim of Pruned-Cascade-Correlation (PCC) is to minimize the *expected* test set error, instead of the actual training error [Weh94]. PCC tries to determine the optimal number of hidden units and to remove unneeded weights after a new hidden unit is installed. As pointed out by Wehrfritz, selection criteria or a hold-out set, as it is used in "stopped-learning", may be applied to digest away unneeded weights. In this release of SNNS, however, only selection criteria for linear models are implemented.

The algorithm works as follows (CC steps are printed italic):

1. *Train the connections to the output layer*
2. Compute the selection criterion
3. *Train the candidates*
4. *Install the new hidden neuron*
5. Compute the selection criterion
6. Set each weight of the last inserted unit to zero and compute the selection criterion; if there exists a weight, whose removal would decrease the selection criterion, remove the link, which decreases the selection criterion **most**. Goto step 5 until a further removal would increase the selection criterion.
7. Compute the selection criterion; if it is greater than the one, computed before inserting the new hidden unit, notify the user that the net is getting too big.

9.8.2.2 Mathematical Background

In this release of SNNS, three model selection criteria are implemented: the Schwarz's Bayesian criterion (SBC), Akaikes information criterion (AIC) and the conservative mean square error of prediction (CMSEP). The *SBC*, the default criterion, is more conservative compared to the *AIC*. Thus, pruning via the *SBC* will produce smaller networks than pruning via the *AIC*. Be aware that both *SBC* and *AIC* are selection criteria for *linear* models, whereas the *CMSEP* does not rely on any statistical theory, but happens to work pretty well in an application. These selection criteria for linear model can sometimes directly be applied to nonlinear models, if the sample size is large.

9.8.3 Recurrent Cascade-Correlation (RCC)

9.8.3.1 The Algorithm

Recurrent Cascade-Correlation (RCC) is a recurrent version of Cascade-Correlation and can be used to train recurrent neural nets ([Elm90]).

Recurrent nets have some features that distinguish them from normal neural networks. For example they can be used to represent time implicitly rather than explicitly. One of the most commonly known architectures of recurrent neural nets is the Elman model, which assumes that the network operates in discrete time steps. The outputs of the network's hidden units at a time t are fed back for use as additional network inputs at time $t+1$. To store the output of the hidden units Elman introduced *context units*, which represent a kind of short-term memory (see fig. 9.3). To integrate the Elman model into the cascade architecture some changes are necessary: The hidden units' values are no longer fed back to all other hidden units. Instead every hidden unit has only one self recurrent link as shown in figure 9.4. This self recurrent link is trained along with the candidate unit's other input weights to maximize the correlation. When the candidate unit is added to the active network as hidden unit, the recurrent link is frozen along with all other links.

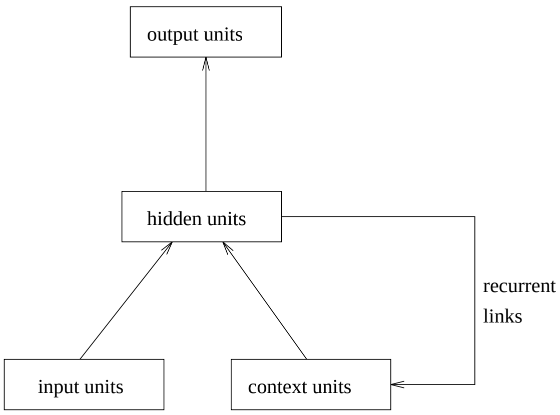

Figure 9.3: The Elman architecture of a recurrent neural net

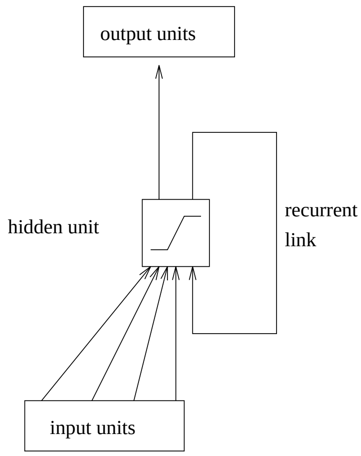

Figure 9.4: RCC architecture of a recurrent neural net.

9.8.3.2 Mathematical Background

The output of a recurrent unit r is given by:

$$V(t) \quad = \quad f_{act}\left(\sum_i I_i(t)w_{ir} + V(t-1)w_s\right),$$

where:
f_{act}: any activation function
w_{ir}: the weight of the connection between the input unit i and the unit r
w_s: the value of the self recurrent connection.
I_i: output of the input unit i

The derivatives of $V(t)$ with respect to w_{ir} and w_s are computed as follows:

$$\frac{\partial V}{\partial w_{ir}} = f'_{act}(t)(I_i(t) + w_s \tfrac{\partial V(t-1)}{\partial w_{ir}})$$

and

$$\frac{\partial V}{\partial w_s} = f'_{act}(t)(V(t-1) + w_s \tfrac{\partial V(t-1)}{\partial w_s}).$$

At $t = 0$, it is assumed that the unit's previous value and previous derivatives are all zero.

9.8.4 Using the Cascade Algorithms in SNNS

Networks that make use of the cascade correlation architecture can be created in SNNS in the same way as all other network types. The control of the training phase, however, is moved from the control panel to the special cascade window described below. The control panel is still used to specify the learning parameters, while the text field `CYCLE` does **not** specify as usual the number of learning cycles. This field is used here to specify the maximal number of hidden units to be generated during the learning phase. The number of learning cycles is entered in the cascade window. The learning parameters for the embedded learning functions Quickprop, Rprop and Backprop are described in chapter 4.3.

If the topology of a net is specified correctly, the program will automatically order the units and layers from left to right in the following way: input layer, hidden layer, output layer, and a candidate layer. [4] The hidden layer is generated with 5 units always having the same x-coordinate (i.e. above each other on the display).

The cascade correlation control panel and the cascade window (see fig. 9.5), is opened by clicking the `Cascade` button in the manager panel. The cascade window is needed to set the parameters of the learning algorithms CC and RCC. To start Cascade Correlation, learning function `CC`, update function `CC_Order` and init function `CC_Weights` in the corresponding menus have to be selected. Recurrent Cascade-Correlation is started in the same way, only that this time the functions `RCC`, `RCC_Update` and `RCC_Weights` have to be selected. If one of these functions is left out, a confirmer window with an error message pops up and learning does not start. The init functions of cascade differ from the normal init functions: upon initialization of a cascade net all hidden units are deleted.

The cascade window has the following text fields, buttons and menus:

[4] The candidate units are realized as special units in SNNS.

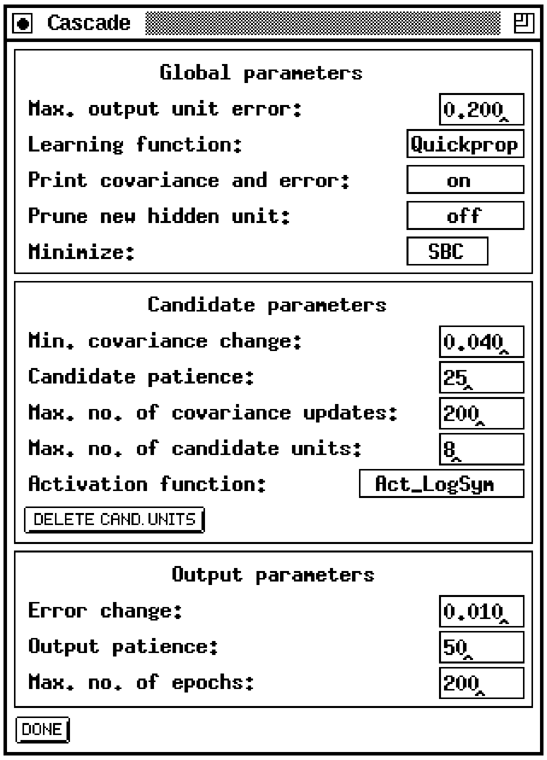

Figure 9.5: The cascade window

- `Global parameters:`
  - `Max. output unit error:`
    This value is used as abort condition for the learning algorithms CC and RCC. If the error of every single output unit is smaller than the given value learning will be terminated.
  - `Learning function:`
    Here, the learning function used to maximize the covariance or to minimize the net error can be selected from a pull down menu. Available learning functions are: `Quickprop`, `Rprop` and `Backprop`
  - `Print covariance and error:`
    If this menu item shows `on`, the development of the error and and the covariance of every candidate unit is printed. `off` prevents all outputs of the net.
  - `Prune new hidden unit:`
    This enables "Pruned-Cascade-Correlation". It defaults to `off`, which means do not remove any weights from the new inserted hidden unit.
  - `Minimize:`
    The selection criterion according to which PCC tries to minimize. The default selection criterion is the "Schwarz's Bayesian criterion", other criteria available are "Akaikes information criterion" and the "conservative mean square error of

prediction". This option is ignored, unless PCC is enabled.

- `Candidate Parameters:`
  - `Min. covariance change:`
    The covariance must change by at least this fraction of its old value to count as a significant change. If this fraction is not reached, learning is halted and the candidate unit with the maximum covariance is changed into a hidden unit.
  - `Candidate patience:`
    After this number of steps the program tests whether there is a significant change of the covariance. The change is said to be significant if it is larger than the fraction given by `Min. covariance change`.
  - `Max. no. of covariance updates:`
    The maximum number of steps to calculate the covariance. After reaching this number, the candidate unit with the maximum covariance is changed to a hidden unit.
  - `Max. no. of candidate units:`
    The maximum number of candidate units trained at once.
  - `Activation function:`
    This menu item makes it possible to choose between different activation functions for the candidate units. The functions are: `Act_Logistic Act_LogSym, Act_Tanh, Act_Identity` and `Random`. `Random` is not a real activation function. It randomly assigns one of the other activation functions to each candidate unit. The function `Act_LogSym` is identical to `Act_Logistic`, except that it is shifted by -0.5 along the y-axis.
- `Output Parameters:`
  - `Error change:`
    analogous to `Min. covariance change`
  - `Output patience:`
    analogous to `Candidate patience`
  - `Max. no. of epochs:`
    analogous to `Max. no. of covariance updates`

The button $\boxed{\text{DELETE CAND. UNITS}}$ deletes all candidate units. In SNNS the candidate units are realized as special units.

9.9 Time Delay Networks (TDNNs)

9.9.1 TDNN Fundamentals

Time delay networks (or TDNN for short), introduced by Alex Waibel ([WHH+89]), are a group of neural networks that have a special topology. They are used for position independent recognition of features within a larger pattern. A special convention for naming different parts of the network is used here (see figure 9.6)

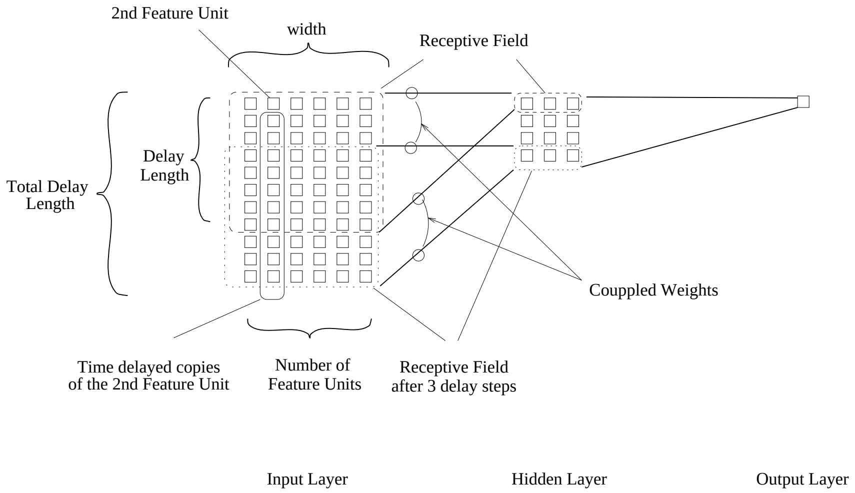

Figure 9.6: The naming conventions of TDNNs

- **Feature:** A component of the pattern to be learned.
- **Feature Unit:** The unit connected with the feature to be learned. There are as many feature units in the input layer of a TDNN as there are features.
- **Delay:** In order to be able to recognize patterns place or time-invariant, older activation and connection values of the feature units have to be stored. This is performed by making a copy of the feature units with all their outgoing connections in each time step, before updating the original units. The total number of time steps saved by this procedure is called *delay*.
- **Receptive Field:** The feature units and their delays are fully connected to the original units of the subsequent layer. These units are called *receptive field*. The receptive field is usually, but not necessarily, as wide as the number of feature units; the feature units might also be split up between several receptive fields. Receptive fields may overlap in the source plane, but do have to cover all feature units.
- **Total Delay Length:** The length of the layer. It equals the sum of the length of all delays of the network layers topological following the current one minus the number of these subsequent layers.
- **Coupled Links:** Each link in a receptive field is reduplicated for every subsequent step of time up to the total delay length. During the learning phase, these links are treated as a single one and are changed according to the average of the changes they would experience if treated separately. Also the units' bias which realizes a special sort of link weight is duplicated over all delay steps of a current feature unit. In figure 9.6 only two pairs of coupled links are depicted (out of 54 quadruples) for simplicity reasons.

The activation of a unit is normally computed by passing the weighted sum of its inputs to an activation function, usually a threshold or sigmoid function. For TDNNs this behavior is modified through the introduction of delays. Now all the inputs of a unit are each multiplied by the N delay steps defined for this layer. So a hidden unit in figure 9.6 would get 6 undelayed input links from the six feature units, and 7x6 = 48 input links from the seven delay steps of the 6 feature units for a total of 54 input connections. Note, that all units in the hidden layer have 54 input links, but only those hidden units activated at time 0 (at the top most row of the layer) have connections to the actual feature units. All other hidden units have the same connection pattern, but shifted to the bottom (i.e. to a later point in time) according to their position in the layer (i.e. delay position in time). By building a whole network of time delay layers, the TDNN can relate inputs in different points in time or input space.

Training in this kind of network is performed by a procedure similar to backpropagation, that takes the special semantics of coupled links into account. To enable the network to achieve the desired behavior, a sequence of patterns has to be presented to the input layer with the feature shifted within the patterns. Remember that since each of the feature units is duplicated for each frame shift in time, the whole history of activations is available at once. But since the shifted copies of the units are mere duplicates looking for the same event, weights of the corresponding connections between the time shifted copies have to be treated as one. First, a regular forward pass of backpropagation is performed, and the error in the output layer is computed. Then the error derivatives are computed and propagated backward. This yields different correction values for corresponding connections. Now all correction values for corresponding links are averaged and the weights are updated with this value.

This update algorithm forces the network to train on time/position independent detection of sub-patterns. This important feature of TDNNs makes them independent from error-prone preprocessing algorithms for time alignment. The drawback is, of course, a rather long, computationally intensive, learning phase.

9.9.2 TDNN Implementation in SNNS

The original time delay algorithm was slightly modified for implementation in SNNS, since it requires either variable network sizes or fixed length input patterns. Time delay networks in SNNS are allowed no delay in the output layer. This has the following consequences:

- The input layer has fixed size.
- Not the whole pattern is present at the input layer at once. Therefore one pass through the network is not enough to compute all necessary weight changes. This makes learning more computationally intensive.

The coupled links are implemented as one physical (i.e. normal) link and a set of logical links associated with it. Only the physical links are displayed in the graphical user interface. The bias of all delay units has no effect. Instead, the bias of the corresponding feature unit is used during propagation and backpropagation.

Activation Function

For time delay networks the new activation function `Act_TD_Logistic` has been implemented. It is similar to the regular logistic activation function `Act_Logistic` but takes care of the special coupled links. The mathematical notation is again

$$a_j(t+1) = \frac{1}{1 + e^{-(\sum_i w_{ij} o_i(t) - \theta_j)}}$$

where o_i includes now also the predecessor units along logical links.

Update Function

The update function `TimeDelay_Order` is used to propagate patterns through a time delay network. It's behavior is analogous to the `Topological_Order` function with recognition of logical links.

Learning Function

The learning function `TimeDelayBackprop` implements the modified backpropagation algorithm discussed above. It uses the same learning parameters as standard backpropagation.

9.9.3 Building and Using a Time Delay Network

In SNNS, TDNNs should be generated only with the tool `BIGNET (Time Delay)`. This program automatically defines the necessary variables and link structures of TDNNs. The logical links are not depicted in the displays and can not be modified with the graphical editor. Any modifications of the units after the creation of the network may result in undesired behavior or even system failure!

After the creation of the net, the unit activation function `Act_TD_Logistic`, the update function `TimeDelay_Order`, and the learning function `TimeDelayBackprop` have to be assigned in the usual way.

NOTE: Only after the special time delay learning function has been assigned, will a save of the network also save the special logical links! A network saved beforehand will lack these links and be useless after a later load operation. Also using the TEST and STEP button will destroy the special time delay information unless the right update function (`TimeDelay_Order`) has been chosen.

Patterns must fit the input layer. If the application requires variable pattern length, a tool to segment these patterns into fitting pieces has to be applied. Patterns may also be generated with the graphical user interface. In this case, it is the responsibility of the user to supply enough patterns with time shifted features for the same teaching output to allow a successful training.

9.10 Radial Basis Functions (RBFs)

The following section describes the use of generalized radial basis functions inside SNNS. First, a brief introduction to the mathematical background of radial basis functions is given. Second, the special procedures of initialization and training of neural nets based on radial basis functions are described. At the end of the chapter a set of necessary actions to use radial basis functions with a specific application are given.

9.10.1 RBF Fundamentals

The principle of radial basis functions derives from the theory of functional approximation. Given N pairs $(\vec{x_i}, y_i)$ $(\vec{x} \in \Re^n, y \in \Re)$ we are looking for a function f of the form:

$$f(\vec{x}) = \sum_{i=1}^{K} c_i h(|\vec{x} - \vec{t_i}|)$$

h is the radial basis function and $\vec{t_i}$ are the K centers which have to be selected. The coefficients c_i are also unknown at the moment and have to be computed. $\vec{x_i}$ and $\vec{t_i}$ are elements of an n–dimensional vector space.

h is applied to the Euclidian distance between each center $\vec{t_i}$ and the given argument $\vec{x}$. Usually a function h which has its maximum at a distance of zero is used, most often the Gaussian function. In this case, values of $\vec{x}$ which are equal to a center $\vec{t}$ yield an output value of 1.0 for the function h, while the output becomes almost zero for larger distances.

The function f should be an approximation of the N given pairs $(\vec{x_i}, y_i)$ and should therefore minimize the following error function H:

$$H[f] = \sum_{i=1}^{N} (y_i - f(\vec{x_i}))^2 + \lambda \|Pf\|^2$$

The first part of the definition of H (the sum) is the condition which minimizes the total error of the approximation, i.e. which constrains f to approximate the N given points. The second part of H ($\|Pf\|^2$) is a stabilizer which forces f to become as smooth as possible. The factor λ determines the influence of the stabilizer.

Under certain conditions it is possible to show that a set of coefficients c_i can be calculated so that H becomes minimal. This calculation depends on the centers $\vec{t_i}$ which have to be chosen beforehand.

Introducing the following vectors and matrices $\vec{c} = (c_1, \cdots, c_K)^T, \vec{y} = (y_1, \cdots, y_N)^T$

$$G = \begin{pmatrix} h(|\vec{x_1} - \vec{t_1}|) & \cdots & h(|\vec{x_1} - \vec{t_K}|) \\ \vdots & \ddots & \vdots \\ h(|\vec{x_N} - \vec{t_1}|) & \cdots & h(|\vec{x_N} - \vec{t_K}|) \end{pmatrix}, \quad G_\square = \begin{pmatrix} h(|\vec{t_1} - \vec{t_1}|) & \cdots & h(|\vec{t_1} - \vec{t_K}|) \\ \vdots & \ddots & \vdots \\ h(|\vec{t_K} - \vec{t_1}|) & \cdots & h(|\vec{t_K} - \vec{t_K}|) \end{pmatrix}$$

the set of unknown parameters c_i can be calculated by the formula:

$$\vec{c} = (G^T \cdot G + \lambda G_{\Box})^{-1} \cdot G^T \cdot \vec{y}$$

By setting λ to 0 this formula becomes identical to the computation of the Moore Penrose inverse matrix, which gives the best solution of an under-determined system of linear equations. In this case, the linear system is exactly the one which follows directly from the conditions of an exact interpolation of the given problem:

$$f(\vec{x_j}) = \sum_{i=1}^{K} c_i h(|\vec{x_j} - \vec{t_i}|) \stackrel{!}{=} y_j \quad , \quad j = 1, \ldots N$$

The method of radial basis functions can easily be represented by a three layer feedforward neural network. The input layer consists of n units which represent the elements of the vector $\vec{x}$. The K components of the sum in the definition of f are represented by the units of the hidden layer. The links between input and hidden layer contain the elements of the vectors $\vec{t_i}$. The hidden units compute the Euclidian distance between the input pattern and the vector which is represented by the links leading to this unit. The activation of the hidden units is computed by applying the Euclidian distance to the function h. Figure 9.7 shows the architecture of the special form of hidden units.

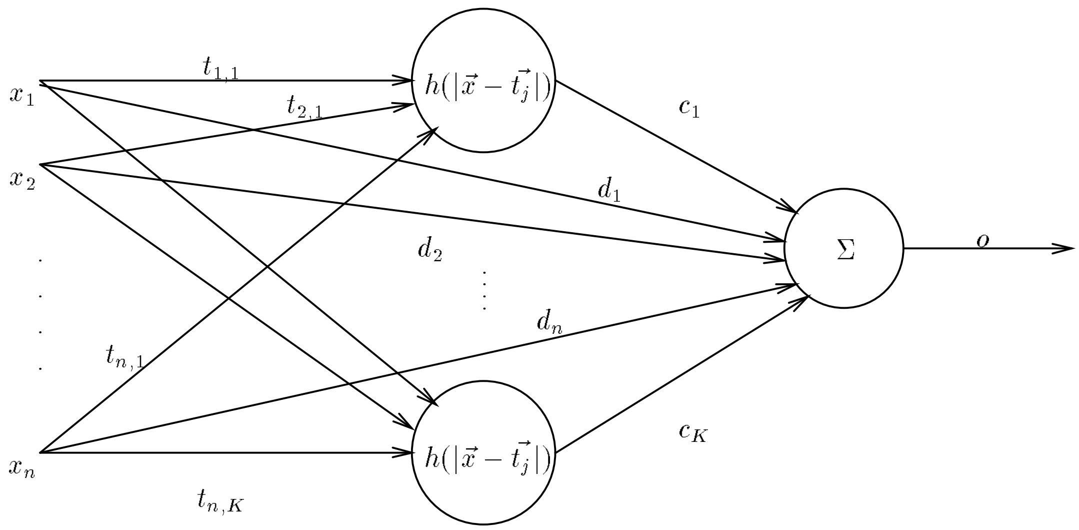

Figure 9.7: The special radial basis unit

The single output neuron gets its input from all hidden neurons. The links leading to the output neuron hold the coefficients c_i. The activation of the output neuron is determined by the weighted sum of its inputs.

The previously described architecture of a neural net, which realizes an approximation using radial basis functions, can easily be expanded with some useful features: More than one output neuron is possible which allows the approximation of several functions f around the same set of centers $\vec{t_i}$. The activation of the output units can be calculated by using a nonlinear invertible function σ (e.g. sigmoid). The bias of the output neurons and a direct connection between input and hidden layer (shortcut connections) can be used to improve the approximation quality. The bias of the hidden units can be used to modify the characteristics of the function h. All in all a neural network is able to represent the following set of approximations:

$$o_k(\vec{x}) = \sigma\left(\sum_{j=1}^{K} c_{j,k} h\left(|\vec{x} - \vec{t_j}|, p_j\right) + \sum_{i=1}^{n} d_{i,k} x_i + b_k\right) = \sigma\left(f_k(\vec{x})\right) , \quad k = 1, \ldots, m$$

This formula describes the behavior of a fully connected feedforward net with n input, K hidden and m output neurons. $o_k(\vec{x})$ is the activation of output neuron k on the input $\vec{x} = x_1, x_2, \ldots, x_n$ to the input units. The coefficients $c_{j,k}$ represent the links between hidden and output layer. The shortcut connections from input to output are realized by $d_{i,k}$. b_k is the bias of the output units and p_j is the bias of the hidden neurons which determines the exact characteristics of the function h. The activation function of the output neurons is represented by σ.

The big advantage of the method of radial basis functions is the possibility of a direct computation of the coefficients $c_{j,k}$ (i.e. the links between hidden and output layer) and the bias b_k. This computation requires a suitable choice of centers $\vec{t_j}$ (i.e. the links between input and hidden layer). Because of the lack of knowledge about the quality of the $\vec{t_j}$, it is recommended to append some cycles of network training after the direct computation of the weights. Since the weights of the links leading from the input to the output layer can also not be computed directly, there must be a special training procedure for neural networks that uses radial basis functions.

The implemented training procedure tries to minimize the error E by using gradient descent. It is recommended to use different learning rates for different groups of trainable parameters. The following set of formulas contains all information needed by the training procedure:

$$E = \sum_{k=1}^{m}\left(\sum_{i=1}^{N}(y_{i,k} - o_k(\vec{x_i}))^2\right) \quad , \quad \Delta\vec{t_j} = -\eta_1 \frac{\partial E}{\partial \vec{t_j}} \quad , \quad \Delta p_j = -\eta_2 \frac{\partial E}{\partial p_j}$$

$$\Delta c_{j,k} = -\eta_3 \frac{\partial E}{\partial c_{j,k}} \quad , \quad \Delta d_{i,k} = -\eta_3 \frac{\partial E}{\partial d_{i,k}} \quad , \quad \Delta b_k = -\eta_3 \frac{\partial E}{\partial b_k}$$

It is often helpful to use a momentum term. This term increases the learning rate in smooth error planes and decreases it in rough error planes. The next formula describes the effect of a momentum term on the training of a general parameter g depending on the additional parameter μ. Δg_{t+1} is the change of g during the time step $t+1$ while Δg_t is the change during time step t:

$$\Delta g_{t+1} = -\eta \frac{\partial E}{\partial g} + \mu \Delta g_t$$

Another useful improvement of the training procedure is the definition of a maximum allowed error inside the output neurons. This prevents the network from getting over-trained, since errors that are smaller than the predefined value are treated as zero. This in turn prevents the corresponding links from being changed.

9.10.2 RBF Implementation in SNNS

9.10.2.1 Activation Functions

For the use of radial basis functions, three different activation functions h have been implemented. For computational efficiency the square of the distance $r^2 = |\vec{x} - \vec{t}|^2$ is uniformly used as argument for h. Also, an additional argument p has been defined which represents the bias of the hidden units. The vectors $\vec{x}$ and $\vec{t}$ result from the activation and weights of links leading to the corresponding unit. The following radial basis functions have been implemented:

1. `Act_RBF_Gaussian` — the Gaussian function

$$h(r^2, p) = h(q, p) = e^{-pq} \quad \text{where} \quad q = |\vec{x} - \vec{t}|^2$$

2. `Act_RBF_MultiQuadratic` — the multiquadratic function

$$h(r^2, p) = h(q, p) = \sqrt{p + q} \quad \text{where} \quad q = |\vec{x} - \vec{t}|^2$$

3. `Act_RBF_ThinPlateSpline` – the *thin plate splines* function

$$\begin{aligned} h(r^2, p) = h(q, p) &= p^2 q \ln(p\sqrt{q}) \quad \text{where} \quad q = |\vec{x} - \vec{t}|^2 \\ &= (pr)^2 \ln(pr) \quad \text{where} \quad r = |\vec{x} - \vec{t}| \end{aligned}$$

During the construction of three layered neural networks based on radial basis functions, it is important to use the three activation functions mentioned above only for neurons inside the hidden layer. There is also only one hidden layer allowed.

For the output layer two other activation functions are to be used:

1. `Act_IdentityPlusBias`
2. `Act_Logistic`

`Act_IdentityPlusBias` activates the corresponding unit with the weighted sum of all incoming activations and adds the bias of the unit. `Act_Logistic` applies the sigmoid logistic function to the weighted sum which is computed like in `Act_IdentityPlusBias`.

In general, it is necessary to use an activation function which pays attention to the bias of the unit.

The last two activation functions converge towards infinity, the first converges towards zero. However, all three functions are useful as base functions. The mathematical preconditions for their use are fulfilled by all three functions and their use is backed by practical experience. All three functions have been implemented as base functions into SNNS.

The most frequently used base function is the Gaussian function. For large distances r, the Gaussian function becomes almost 0. Therefore, the behavior of the net is easy to predict if the input patterns differ strongly from all teaching patterns. Another advantage of the Gaussian function is, that the network is able to produce useful results without the use of shortcut connections between input and output layer.

9.10.2.2 Initialization Functions

The goal in initializing a radial basis function network is the optimal computation of link weights between hidden and output layer. Here the problem arises that the centers $\vec{t_j}$ (i.e. link weights between input and hidden layer) as well as the parameter p (i.e. the bias of the hidden units) must be set properly. Therefore, three different initialization procedures have been implemented which perform different tasks:

1. `RBF_Weights`: This procedure first selects evenly distributed centers $\vec{t_j}$ from the loaded training patterns and assigns them to the links between input and hidden layer. Subsequently the bias of all neurons (parameter p) inside the hidden layer is set to a value determined by the user and finally the links between hidden and output layer are computed.
2. `RBF_Weights_Redo`: In contrast to the preceding procedure only the links between hidden and output layer are computed. All other links and bias remain unchanged.
3. `RBF_Weights_Kohonen`: Using the self–organizing method of Kohonen feature maps, appropriate centers are generated on base of the teaching patterns. The computed centers are copied into the corresponding links. No other links and bias are changed.

It is necessary that valid patterns are loaded into SNNS to use the initialization. If no patterns are present upon starting any of the three procedures an alert box will occur showing the error. A detailed description of the procedures and the parameters used is given in the following paragraphs.

RBF_Weights Of the named three procedures `RBF_Weights` is the most comprehensive one. Here all necessary initialization tasks (setting link weights and bias) for a fully connected three layer feedforward network (without shortcut connections) can be performed in one single step. Hence, the choice of centers (i.e. the link weights between input and hidden layer) is rather simple: The centers are evenly selected from the loaded teaching patterns and assigned to the links of the hidden neurons. The selection process assigns the first teaching pattern to the first hidden unit, and the last pattern to the last hidden unit. The remaining hidden units receive centers which are evenly picked from the set of teaching patterns. If, for example, 13 teaching patterns are loaded and the hidden layer consists of 5 neurons, then the patterns with numbers 1, 4, 7, 10 and 13 are selected as centers.

Before a selected teaching pattern is distributed among the corresponding link weights it can be modified slightly with a random number. For this purpose, an initialization parameter (*deviation*, parameter 5) is set, which determines the maximum percentage of deviation allowed to occur randomly. To calculate the deviation, an inverse tangent function is used to approximate a normal distribution so that small deviations are more probable than large deviations. Setting the parameter *deviation* to 1.0 results in a maximum deviation of 100%. The centers are copied unchanged into the link weights if the deviation is set to 0.

A small modification of the centers is recommended for the following reasons: First, the number of hidden units may exceed the number of teaching patterns. In this case it is necessary to break the symmetry which would result without modification. This symmetry would render the calculation of the Moore Penrose inverse matrix impossible. The second reason is that there may be a few anomalous patterns inside the set of teaching patterns. These patterns would cause bad initialization results if they accidentally were selected as a center. By adding a small amount of noise, the negative effect caused by anomalous patterns can be lowered. However, if an exact interpolation is to be performed no modification of centers may be allowed.

The next initialization step is to set the free parameter p of the base function h, i.e. the bias of the hidden neurons. In order to do this, the initialization parameter *bias (p)* is directly copied into the bias of all hidden neurons. The setting of the bias is highly related to the base function h used and to the properties of the teaching patterns. When the Gaussian function is used, it is recommended to choose the value of the bias so that 5–10% of all hidden neurons are activated during propagation of every single teaching pattern. If the bias is chosen too small, almost all hidden neurons are uniformly activated during propagation. If the bias is chosen too large, only that hidden neuron is activated whose center vector corresponds to the currently applied teaching pattern.

Now the expensive initialization of the links between hidden and output layer is actually performed. In order to do this, the following formula which was already presented above is applied:

$$\vec{c} = (G^T \cdot G + \lambda G_{\square})^{-1} \cdot G^T \cdot \vec{y}$$

The initialization parameter 3 (*smoothness*) represents the value of λ in this formula. The matrices have been extended to allow an automatic computation of an additional constant value. If there is more than one neuron inside the output layer, the following set of functions results:

$$f_j(\vec{x}) = \sum_{i=1}^{K} c_{i,j} h_i(\vec{x}) + b_j$$

The bias of the output neuron(s) is directly set to the calculated value of b (b_j). Therefore, it is necessary to choose an activation function for the output neurons that uses the bias of the neurons. In the current version of SNNS, the functions `Act_Logistic` and `Act_IdentityPlusBias` implement this feature.

The activation functions of the output units lead to the remaining two initialization parameters. The initialization procedure assumes a linear activation of the output units. The link weights are calculated so that the weighted sum of the hidden neurons equals the teaching output. However, if a sigmoid activation function is used, which is recommended for pattern recognition tasks, the activation function has to be considered during initialization. Ideally, the supposed input for the activation function should be computed with the inverse activation function depending on the corresponding teaching output. This input value would be associated with the vector $\vec{y}$ during the calculation of weights. Unfortunately, the inverse activation function is unknown in the general case.

The first and second initialization parameters (*0_scale*) and (*1_scale*) are a remedy for this dilemma. They define the two control points of a piecewise linear function which approximates the activation function. *0_scale* and *1_ scale* give the net inputs of the output units which produce the teaching outputs 0 and 1. If, for example, the linear activation function `Act_IdentityPlusBias` is used, the values 0 and 1 have to be used. When using the logistic activation function `Act_Logistic`, the values -4 and 4 are recommended. If the bias is set to 0, these values lead to a final activation of 0.018 (resp. 0.982). These are comparatively good approximations of the desired teaching outputs 0 and 1. The implementation interpolates linearly between the set values of *0_scale* and *1_scale*. Thus, also teaching values which differ from 0 and 1 are mapped to corresponding input values.

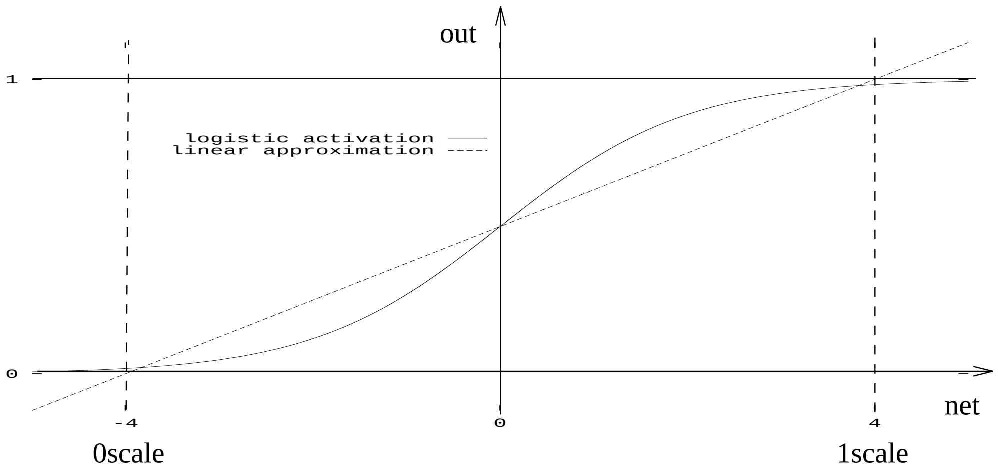

Figure 9.8: Relation between teaching output, input value and logistic activation

Figure 9.8 shows the activation of an output unit under use of the logistic activation function. The scale has been chosen in such a way, that the teaching outputs 0 and 1 are mapped to the input values -2 and 2.

The optimal values used for *0_scale* and *1_scale* can not be given in general. With the logistic activation function large scaling values lead to good initialization results, but interfere with the subsequent training, since the logistic function is used mainly in its very flat parts. On the other hand, small scaling values lead to bad initialization results, but produce good preconditions for additional training.

RBF_Weights_Kohonen One disadvantage of the above initialization procedure is the very simple selection of center vectors from the set of teaching patterns. It would be favorable if the center vectors would homogeneously cover the space of teaching patterns. `RBF_Weights_Kohonen` allows a self–organizing training of center vectors. Here, just as the name of the procedure already tells, the self–organizing maps of Kohonen are used (see [Was89]). The simplest version of Kohonen's maps has been implemented. It works as follows:

One precondition for the use of Kohonen maps is that the teaching patterns have to be normalized. This means, that they represent vectors with length 1. K patterns have to be selected from the set of n teaching patterns acting as starting values for the center vectors. Now the scalar product between one teaching pattern and each center vector is computed. If the vectors are normalized to length 1, the scalar product gives a measure for the distance between the two multiplied vectors. Now the center vector is determined whose distance to the current teaching pattern is minimal, i.e. whose scalar product is the largest one. This center vector is moved a little bit in the direction of the current teaching pattern:

$$\vec{z}_{\text{new}} = \vec{z}_{\text{old}} + \alpha(\vec{l} - \vec{z}_{\text{old}})$$

This procedure is repeated for all teaching patterns several times. As a result, the center vectors adapt the statistical properties of the set of teaching patterns.

The resp. meanings of the three initialization parameters are:

1. *learn cycles*: determines the number of iterations of the Kohonen training for all teaching patterns. If 0 epochs are specified only the center vectors are set, but no training is performed.
2. *learning rate* α: It should be picked between 0 and 1. A learning rate of 0 leaves the center vectors unchanged. Using a learning rate of 1 replaces the selected center vector by the current teaching pattern.
3. *shuffle*: Determines the selection of initial center vectors at the beginning of the procedure. A value of 0 leads to the even selection already described for `RBF_Weights`. Any value other than 0 causes a random selection of center vectors from the set of teaching patterns.

Note, that the described initialization procedure initializes only the center vectors (i.e. the link weights between input and hidden layer). The bias values of the neurons have to be set manually using the graphical user interface. To perform the final initialization of missing link weights, another initialization procedure has been implemented.

RBF_Weights_Redo This initialization procedure influences only the link weights between hidden and output layer. It initializes the network as well as possible by taking the bias and the center vectors of the hidden neurons as a starting point. The center vectors can be set by the previously described initialization procedure. Another possibility is to create the center vectors by an external procedure, convert these center vectors into a SNNS pattern file and copy the patterns into the corresponding link weights by using the previously described initialization procedure. When doing this, Kohonen training must not be performed of course.

The effect of the procedure `RBF_Weights_Redo` differs from `RBF_Weights` only in the way that the center vectors and the bias remain unchanged. As expected, the last two initialization parameters are omitted. The meaning and effect of the remaining three parameters is identical with the ones described in `RBF_Weights`.

9.10.2.3 Learning Functions

Because of the special activation functions used for radial basis functions, a special learning function is needed. It is impossible to train networks which use the activation functions `Act_RBF_`... with backpropagation. The learning function for radial basis functions implemented here can only be applied if the neurons which use the special activation functions are forming the hidden layer of a three layer feedforward network. Also the neurons of the output layer have to pay attention to their bias for activation.

The name of the special learning function is `RadialBasisLearning`. The required parameters are:

1. η_1 *(centers)*: the learning rate used for the modification $\Delta\vec{t_j}$ of center vectors according to the formula $\Delta\vec{t_j} = -\eta_1 \frac{\partial E}{\partial \vec{t_j}}$.
2. η_2 *(bias p)*: learning rate used for the modification of the parameters p of the base function. p is stored as bias of the hidden units and is trained by the following formula $\Delta p_j = -\eta_2 \frac{\partial E}{\partial p_j}$.
3. η_3 *(weights)*: learning rate which influences the training of all link weights that are leading to the output layer as well as the bias of all output neurons.

$$\Delta c_{j,k} = -\eta_3 \frac{\partial E}{\partial c_{j,k}} \quad , \quad \Delta d_{i,k} = -\eta_3 \frac{\partial E}{\partial d_{i,k}} \quad , \quad \Delta b_k = -\eta_3 \frac{\partial E}{\partial b_k}$$

4. *delta max.*: To prevent an overtraining of the network the maximally tolerated error in an output unit can be defined. If the actual error is smaller than *delta max.* the corresponding weights are not changed. Common values range from 0 to 0.3.
5. *momentum*: momentum term during training, after the formula $\Delta g_{t+1} = -\eta \frac{\partial E}{\partial g} + \mu \Delta g_t$. The momentum–term is usually chosen between 0.8 and 0.9.

The learning rates η_1 to η_3 have to be selected very carefully. If the values are chosen too large (like the size of values for backpropagation) the modification of weights will be too extensive and the learning function will become unstable. Tests showed, that the learning procedure becomes more stable if only one of the three learning rates is set to a value bigger than 0. Most critical is the parameter *bias (p)*, because the base functions are fundamentally changed by this parameter.

Tests also showed that the learning function working in batch mode is much more stable than in online mode. Batch mode means that all changes become active not before all learning patterns have been presented once. This is also the training mode which is recommended in the literature about radial basis functions. The opposite of batch mode is known as online mode, where the weights are changed after the presentation of every single teaching pattern. Which mode is to be used can be defined during compilation of SNNS. The online mode is activated by defining the C macro `RBF_INCR_LEARNING` during compilation of the simulator kernel, while batch mode is the default.

9.10.3 Building a Radial Basis Function Application

As a first step, a three–layer feedforward network must be constructed with full connectivity between input and hidden layer and between hidden and output layer. Either the graphical editor or the tool `BIGNET` (both built into SNNS) can be used for this purpose.

The output function of all neurons is set to `Out_Identity`. The activation function of all hidden layer neurons is set to one of the three special activation functions `Act_RBF_`... (preferably to `Act_RBF_Gaussian`). For the activation of the output units, a function is needed which takes the bias into consideration. These functions are `Act_Logistic` and `Act_IdentityPlusBias`.

The next step consists of the creation of teaching patterns. They can be generated manually using the graphical editor, or automatically from external data sets by using an appropriate conversion program. If the initialization procedure `RBF_Weights_Kohonen` is going to be used, the center vectors should be normalized to length 1, or to equal length.

It is necessary to select an appropriate bias for the hidden units before the initialization is continued. Therefore, the link weights between input and hidden layer are set first, using the procedure `RBF_Weights_Kohonen` so that the center vectors which are represented by the link weights form a subset of the available teaching patterns. The necessary initialization parameters are: *learn cycles* = 0, *learning rate* = 0.0, *shuffle* = 0.0. Thereby teaching patterns are used as center vectors without modification.

To set the bias, the activation of the hidden units is checked for different teaching patterns by using the button TEST of the SNNS control panel. When doing this, the bias of the hidden neurons have to be adjusted so that the activations of the hidden units are as diverse as possible. Using the Gaussian function as base function, all hidden units are uniformly highly activated, if the bias is chosen too small (the case bias = 0 leads to an activation of 1 of all hidden neurons). If the bias is chosen too large, only the unit is activated whose link weights correspond to the current teaching pattern. A useful procedure to find the right bias is to first set the bias to 1, and then to change it uniformly depending on the behavior of the network. One must take care, however, that the bias does not become negative, since some implemented base functions require the bias to be positive. The optimal choice of the bias depends on the dimension of the input layer and the similarity among the teaching patterns.

After a suitable bias for the hidden units has been determined, the initialization procedure `RBF_Weights` can be started. Depending on the selected activation function for the output layer, the two *scale* parameters have to be set (see page 146). When `Act_IdentityPlusBias` is used, the two values 0 and 1 should be chosen. For the logistic activation function `Act_Logistic` the values -4 and 4 are recommended (also see figure 9.8). The parameters *smoothness* and *deviation* should be set to 0 first. The *bias* is set to the previously determined value. Depending on the number of teaching patterns and the number of hidden neurons, the initialization procedure may take rather long to execute. Therefore, some processing comments are printed on the terminal during initialization.

After the initialization has finished, the result may be checked by using the TEST button. However, the exact network error can only be determined by the teaching function. Therefore, the learning function `RadialBasisLearning` has to be selected first. All learning parameters are set to 0 and the number of learning cycles (CYCLES) is set to 1. After pressing the button ALL, the learning function is started. Since the learning parameters are set to 0, no changes inside the network will occur. After the presentation of all available teaching patterns, the actual error is printed to the terminal. As usual, the error is defined as the sum of squared errors of all output units (see formula 9.3). Under certain conditions it can be possible that the error becomes very large. This is mostly due to numerical problems. A poorly selected bias, for example, has shown to be a difficult starting point for the initialization. Also, if the number of teaching patterns is less than or equal to the number of hidden units a problem arises. In this case the number of unknown weights plus unknown bias values of output units exceeds the number of teaching patterns, i.e. there are more unknown parameters to be calculated than equations available. One or more neurons less inside the hidden layer then reduces the error considerably.

After the first initialization it is recommended to save the current network to test the possibilities of the learning function. It has turned out that the learning function becomes quickly unstable if too large learning rates are used. It is recommended to first set only one of the three learning rates (*centers, bias (p), weights*) to a value larger than 0 and to check the sensitivity of the learning function on this single learning rate. The use of the parameter *bias (p)* is exceptionally critical because it causes serious changes of the base function. If the bias of any hidden neuron is getting negative during learning, an appropriate message is printed to the terminal. In that case, a continuing meaningful training is impossible and the network should be reinitialized.

Immediately after initialization it is often useful to train only the link weights between hidden and output layer. Thereby the numerical inaccuracies which appeared during initialization are corrected. However, an optimized total result can only be achieved if also the center vectors are trained, since they might have been selected disadvantageously.

The initialization procedure used for direct link weight calculation is unable to calculate the weights between input and output layer. If such links are present, the following procedure is recommended: Even before setting the center vectors by using `RBF_Weights_Kohonen`, and before searching an appropriate bias, all weights should be set to random values between -0.1 and 0.1 by using the initialization procedure `Randomize_Weights`. Thereby, all links between input and output layer are preinitialized. Later on, after executing the procedure `RBF_Weights`, the error of the network will still be relatively large, because the above mentioned links have not been considered. Now it is easy to train these weights by only using the teaching parameter *weights* during learning.

9.11 Dynamic Decay Adjustment for RBFs (RBF–DDA)

9.11.1 The Dynamic Decay Adjustment Algorithm

The Dynamic Decay Adjustment (DDA–)Algorithm is an extension of the RCE-Algorithm (see [Hud92, RCE82]) and offers easy and constructive training for Radial Basis Function Networks. RBFs trained with the DDA-Algorithm often achieve classification accuracy comparable to Multi Layer Perceptrons (MLPs)[5] but training is significantly faster ([BD95]).

An RBF trained with the DDA-Algorithm (RBF-DDA) is similar in structure to the common feedforward MLP with one hidden layer and without shortcut connections:

- The number of units in the **input layer** represents the dimensionality of the input space.
- The **hidden layer** contains the RBF units. Units are added in this layer during training. The input layer is fully connected to the hidden layer.
- Each unit in the **output layer** represents one possible class, resulting in an 1-of-n or binary coding. For classification a winner-takes-all approach is used, i.e. the output with the highest activation determines the class. Each hidden unit is connected to exactly one output unit.

The main differences to an MLP are the activation function and propagation rule of the hidden layer: Instead of using a sigmoid or another nonlinear squashing function, RBFs use localized functions, radial Gaussians, as an activation function. In addition, a computation of the Euclidian distance to an individual reference vector replaces the scalar product used in MLPs:

$$R_i(\vec{x}) = exp(-\frac{||\vec{x} - \vec{r}_i||^2}{\sigma_i^2})$$

If the network receives vector $\vec{x}$ as an input, R_i indicates the activation of one RBF unit with reference vector $\vec{r}_i$ and standard deviation σ_i.

The output layer computes the output for each class as follows:

$$f(\vec{x}) = \sum_{i=1}^{m} A_i * R_i(\vec{x})$$

with m indicating the number of RBFs belonging to the corresponding class and A_i being the weight for each RBF.

An example of a full RBF-DDA is shown in figure 9.9. Note that there do not exist any shortcut connections between input and output units in an RBF-DDA.

In this illustration the weight vector that connects all input units to one hidden unit represents the centre of the Gaussian. The Euclidian distance of the input vector to this reference vector (or *prototype*) is used as an input to the Gaussian which leads to a local

[5] As usual the term MLP refers to a multilayer feedforward network using the scalar product as a propagation rule and sigmoids as transfer functions.

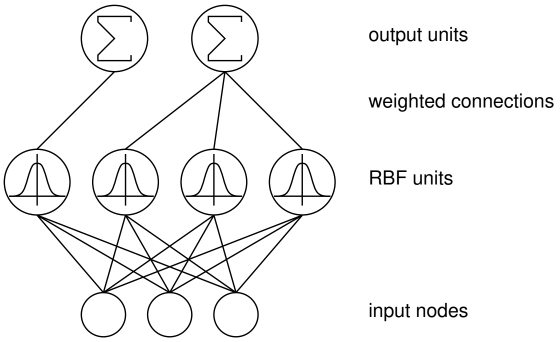

Figure 9.9: The structure of a Radial Basis Function Network.

response; if the input vector is *close* to the prototype, the unit will have a high activation. In contrast the activation will be close to zero for larger distances. Each output unit simply computes a weighted sum of all activations of the RBF units belonging to the corresponding class.

The DDA-Algorithm introduces the idea of distinguishing between *matching* and *conflicting* neighbors in an *area of conflict*. Two thresholds θ^+ and θ^- are introduced as illustrated in figure 9.10.

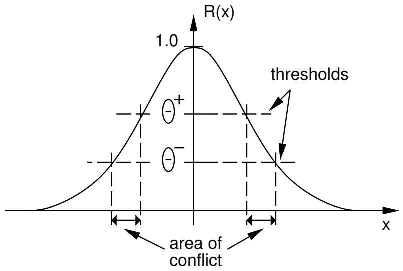

Figure 9.10: One RBF unit as used by the DDA-Algorithm. Two *thresholds* are used to define an *area of conflict* where no other prototype of a conflicting class is allowed to exist. In addition, each training pattern has to be in the inner circle of at least one prototype of the correct class.

Normally, θ^+ is set to be greater than θ^- which leads to a *area of conflict* where neither matching nor conflicting training patterns are allowed to lie[6]. Using these thresholds, the algorithm constructs the network dynamically and adjusts the radii individually.

In short the main properties of the DDA-Algorithm are:

- **constructive training**: new RBF nodes are added whenever necessary. The network is built from scratch, the number of required hidden units is determined during training. Individual radii are adjusted dynamically during training.
- **fast training**: usually about five epochs are needed to complete training, due to the constructive nature of the algorithm. End of training is clearly indicated.

[6]The only exception to this rule is the case where a pattern of the same class lies in the area of conflict but is covered by another RBF (of the correct class) with a sufficiently high activation.

- **guaranteed convergence**: the algorithm can be proven to terminate.
- **two uncritical parameters**: only the two parameters θ^+ and θ^- have to be adjusted manually. Fortunately the values of these two thresholds are not critical to determine. For all tasks that have been used so far, $\theta^+ = 0.4$ and $\theta^- = 0.2$ was a good choice.
- **guaranteed properties** of the network: it can be shown that after training has terminated, the network holds several conditions for all training patterns: wrong classifications are below a certain threshold (θ^-) and correct classifications are above another threshold (θ^+).

The DDA-Algorithm is based on two steps. During training, whenever a pattern is misclassified, either a new RBF unit with an initial weight = 1 is introduced (called *commit*) or the weight of an existing RBF (which covers the new pattern) is incremented. In both cases the radii of conflicting RBFs (RBFs belonging to the wrong class) are reduced (called *shrink*). This guarantees that each of the patterns in the training data is covered by an RBF of the correct class and none of the RBFs of a conflicting class has an inappropriate response.

Two parameters are introduced at this stage, a *positive threshold* θ^+ and a *negative threshold* θ^-. To commit a new prototype, none of the existing RBFs of the correct class has an activation above θ^+ and during shrinking no RBF of a conflicting class is allowed to have an activation above θ^-. Figure 9.11 shows an example that illustrates the first few training steps of the DDA-Algorithm.

After training is finished, two conditions are true for all input–output pairs[7] $(\vec{x}, c)$ of the training data:

- at least one prototype of the correct class c has an activation value greater or equal to θ^+:

$$\exists i : R_i^c(\vec{x}) \geq \theta^+$$

- all prototypes of conflicting classes have activations less or equal to θ^- (m_k indicates the number of prototypes belonging to class k):

$$\forall k \neq c, 1 \leq j \leq m_k : R_j^k(\vec{x}) \leq \theta^-$$

For all experiments conducted so far, the choice of θ^+=0.4 and θ^-=0.2 led to satisfactory results. In theory, those parameters should be dependent on the dimensionality of the feature space but in practice the values of the two thresholds seem to be uncritical. Much more important is that the input data is normalized. Due to the radial nature of RBFs each attribute should be distributed over an equivalent range. Usually normalization into [0,1] is sufficient.

[7] In this case the term "input–class pair" would be more justified, since the DDA–Algorithm trains the network to classify rather than approximate an input–output mapping.

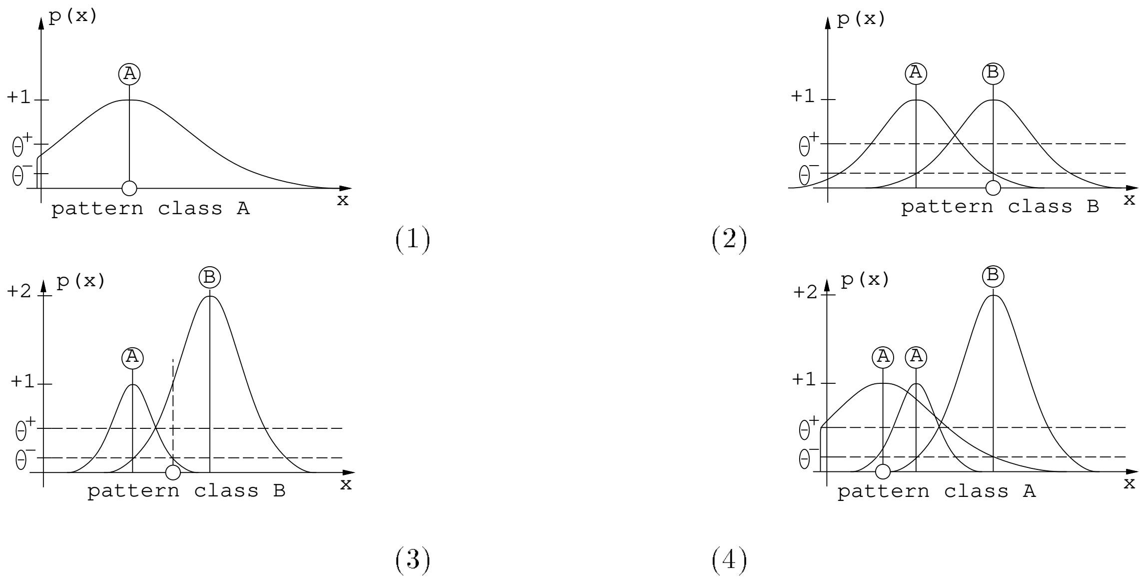

Figure 9.11: An example of the DDA-Algorithm: (1) a pattern of class A is encountered and a new RBF is created; (2) a training pattern of class B leads to a new prototype for class B and shrinks the radius of the existing RBF of class A; (3) another pattern of class B is classified correctly and shrinks again the prototype of class A; (4) a new pattern of class A introduces another prototype of that class.

9.11.2 Using RBF–DDA in SNNS

The implementation of the DDA-Algorithm always uses the Gaussian activation function `Act_RBF_Gaussian` in the hidden layer. All other activation and output functions are set to `Act_Identity` and `Out_Identity`, respectively.

The *Learning function* has to be set to `RBF-DDA`. No *Initialization* or *Update* functions are needed.

The algorithm takes three arguments that are set in the first three fields of the `LEARN` row in the *control panel*. These are θ^+, θ^- $(0 < \theta^- \leq \theta^+ < 1)$ and the maximum number of RBF units to be diplayed in one row. This last item allows the user to control the appearance of the network on the screen and has no influence on the performance. Specifying 0.0 leads to the default values $\theta^+ = 0.4, \theta^- = 0.2$ and to a maximum number of 20 RBF units displayed in a row.

Training of an RBF starts with either:

- an **empty** network, i.e. a network consisting only of input and output units. No connections between input and output units are required, hidden units will be added during training. This can easily be generated with the tool `BIGNET` (choice `FEED FORWARD`)
- a **pretrained** network already containing RBF units generated in an earlier run of

`RBF-DDA` (all networks not complying with the specification of an RBF-DDA will be rejected).

After having loaded a training pattern set, a *learning-epoch* can be started by pressing the `ALL` button in the *control panel*. At the beginning of each epoch, the weights between the hidden and the output layer are automatically set to zero. Note that the resulting RBF and the number of required learning epochs can vary slightly depending on the order of the training patterns. If you train using a single pattern (by pressing the `SINGLE` button) keep in mind that every training step increments the weight between the RBF unit of the correct class covering that pattern and its corresponding output unit. The end of the training is reached when the network structure does not change any more and the Mean Square Error (MSE) stays constant from one epoch to another.

The first desired value in an output pattern that is greater than 0.0 will be assumed to represent the class this pattern belongs to; only one output may be greater than 0.0. If there is no such output, training is still executed, but no new prototype for this pattern is commited. All existing prototypes are shrunk to avoid coverage of this pattern, however. This can be an easy way to define an "error"–class without trying to model the class itself.

9.12 ART Models in SNNS

This section will describe the use of the three ART models ART1, ART2 and ARTMAP, as they are implemented in SNNS. It will not give detailed information on the Adaptive Resonance Theory. You should already know the theory to be able to understand this chapter. For the theory the following literature is recommended:

[CG87a] Original paper, describing ART1 theory.

[CG87b] Original paper, describing ART2 theory.

[CG91] Original paper, describing ARTMAP theory.

[Her92] Description of theory, implementation and application of the ART models in SNNS (in German).

There will be one subsection for each of the three models and one subsection describing the required topologies of the networks when using the ART learning-, update- or initialization-functions. These topologies are rather complex. For this reason the network creation tool **BigNet** has been extended. It now offers an easy way to create ART1, ART2 and ARTMAP networks according to your requirements. For a detailed explanation of the respective features of **BigNet** see chapter 7.

9.12.1 ART1

9.12.1.1 Structure of an ART1 Network

The topology of ART1 networks in SNNS has been chosen to to perform most of the ART1 algorithm within the network itself. This means that the mathematics is realized in the

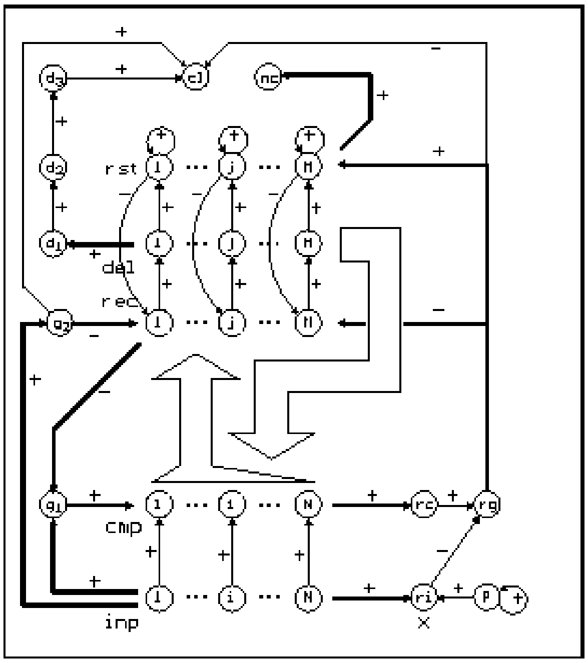

Figure 9.12:
Structure of an ART1 network in SNNS. Thin arrows represent a connection from one unit to another. Fat arrows which go from a layer to a unit indicate that each unit of the layer is connected to the target unit. Similarly a fat arrow from a unit to a layer means that the source unit is connected to each of the units in the target layer. The two big arrows in the middle represent the full connection between comparison and recognition layer and the one between delay and comparison layer, respectively.

activation and output functions of the units. The idea was to keep the propagation and training algorithm as simple as possible and to avoid procedural control components.

In figure 9.12.1.1 the units and links of ART1 networks in SNNS are displayed.

The F_0 or input layer (labeled **inp** in figure 9.12.1.1) is a set of N input units. Each of them has a corresponding unit in the F_1 or comparison layer (labeled **cmp**). The M elements in the F_2 layer are split into three levels. So each F_2 element consists of three units. One recognition (**rec**) unit, one delay (**del**) unit and one local reset (**rst**) unit. These three parts are necessary for different reasons. The recognition units are known from the theory. The delay units are needed to synchronize the network correctly[8]. Besides, the activated unit in the delay layer shows the winner of F_2. The job of the local reset units is to block the actual winner of the recognition layer in case of a reset.

Finally, there are several special units. The **cl** unit gets positive activation when the input pattern has been successfully classified. The **nc** unit indicates an unclassifiable pattern, when active. The gain units $\mathbf{g}_1$ and $\mathbf{g}_2$ with their known functions and at last the units **ri** (reset input), **rc** (reset comparison), **rg** (reset general) and ρ(vigilance), which realize the reset function.

For an exact definition of the required topology for ART1 networks in SNNS see section 9.12.4

[8]This is only important for the chosen realization of the ART1 learning algorithm in SNNS

9.12.1.2 Using ART1 Networks in SNNS

To use an ART1 network in SNNS several functions have been implemented: one to initialize the network, one to train it and two different update functions to propagate an input pattern through the net.

ART1 Initialization Function First the ART1 initialization function `ART1_Weights` has to be selected from the list of initialization functions.

`ART1_Weights` is responsible to set the initial values of the trainable links in an ART1 network. These links are the ones from F_1 to F_2 and the ones from F_2 to F_1 respectively.

The $F_2 \rightarrow F_1$ links are all set to 1.0 as described in [CG87a]. The weights of the links from F_1 to F_2 are a little more difficult to explain. To assure that in an initialized network the F_2 units will be used in their index order, the weights from F_1 to F_2 must decrease with increasing index. Another restriction is, that each link-weight has to be greater than 0 and smaller than $1/N$. Defining α_j as a link-weight from a F_1 unit to the jth F_2 unitthis yields

$$0 < \alpha_M < \alpha_{M-1} < \ldots < \alpha_1 \leq \frac{1}{\beta + N}. \tag{9.3}$$

To get concrete values, we have to decrease the fraction on the right side with increasing index j and assign this value to α_j. For this reason we introduce the value η and we obtain

$$\alpha_j \equiv \frac{1}{\beta + (1 + j\eta)N}. \tag{9.4}$$

η is calculated out of a new parameter γ and the number of F_2 units M:

$$\eta \equiv \frac{\gamma}{M}. \tag{9.5}$$

So we have two parameters for `ART1_Weights`: β and γ. For both of them a value of 1.0 is useful for the initialization. The first parameter of the initialization function is β, the second one is γ. Having chosen β and γ one must press the INIT-button to perform initialization.

The parameter β is stored in the bias field of the unit structure to be accessible to the learning function when adjusting the weights.

One should always use `ART1_Weights` to initialize ART1 networks. When using another SNNS initialization function the behavior of the simulator during learning is not predictable, because not only the trainable links will be initialized, but also the fixed weights of the network.

ART1 Learning Function To train an ART1 network select the learning function `ART1`. To start the training of an ART1 network, choose the vigilance parameter ρ (e.g.: 0.1) as first value in both `LEARN` and `UPDATE` row of the control panel. Parameter β, which is also needed to adjust the trainable weights between F_1 and F_2, has already been specified as initialization parameter. It is stored in the bias field of the unit structure and read out by `ART1` when needed.

ART1 Update Functions To propagate a new pattern through an ART1 network without adjusting weights, i.e. to classify a pattern, two different update functions have been implemented:

- `ART1_Stable` and
- `ART1_Synchronous`.

Like the learning function, both of the update functions only take the vigilance value ρ as parameter. It has to be entered in the control panel, the line below the parameters for the learning function. The difference between the two update functions is the following:

`ART1_Stable` propagates a pattern until the network is stable, i.e. either the **cl** unit or the **nc** unit is active. To use this update function, you can use the TEST-button of the control panel. The next pattern is copied to the input units and propagated completely through the net, until a stable state is reached.

`ART1_Synchronous`, performs just one propagation step with each call. To use this function you have to press the RESET-button to reset the net to a defined initial state, where each unit has its initial activation value. Then copy a new pattern into the input layer, using the buttons $<$ and $>$. Now you can choose the desired number of propagation steps that should be performed, when pressing the STEP-button (default is 1). With this update function it is very easy to observe how the ART1 learning algorithm does its job.

So use `ART1_Synchronous`, to *trace* a pattern through a network, `ART1_Stable` to propagate the pattern until a stable state is reached.

9.12.2 ART2

9.12.2.1 Structure of an ART2 Network

The realization of ART2 differs from the one of ART1 in its basic idea. In this case the network structure would have been too complex, if mathematics had been implemented within the network to the same degree as it has been done for ART1. So here more of the functionality is in the control program. In figure 9.12.2.1 you can see the topology of an ART2 network as it is implemented in SNNS.

All the units are known from the ART2 theory, except the **rst** units. They have to do the same job for ART2 as for ART1 networks. They block the actual winner in the recognition layer in case of reset. Another difference between the ART2 model described in [CG87b]

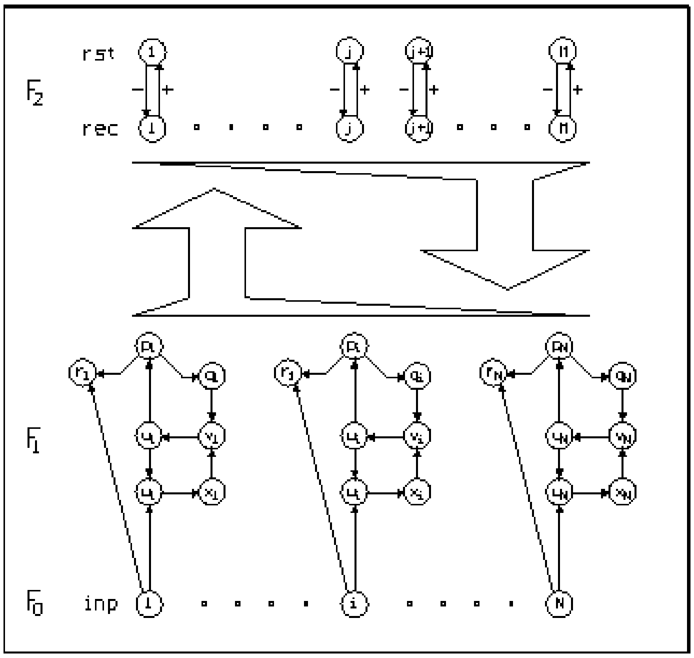

Figure 9.13:
Structure of an ART2 network in SNNS. Thin arrows represent a connection from one unit to another. The two big arrows in the middle represent the full connectivity between comparison and recognition layer and the one between recognition and comparison layer, respectively.

and the realization in SNNS is, that originally the units $\mathbf{u}_i$ have been used to compute the error vector $\mathbf{r}$, while this implementation takes the input units instead.

For an exact definition of the required topology for ART2 networks in SNNS see section 9.12.4

9.12.2.2 Using ART2 Networks in SNNS

As for ART1 there are an initialization function, a learning function and two update functions for ART2. To initialize, train or test an ART2 network, these functions have to be used. The description of the handling, is not repeated in detail in this section since it is the same as with ART1. Only the parameters for the functions will be mentioned here.

ART2 Initialization Function For an ART2 network the weights of the top-down-links ($F_2 \rightarrow F_1$ links) are set to 0.0 according to the theory ([CG87b]).

The choice of the initial bottom-up-weights is determined as follows: if a pattern has been trained, then the next presentation of the same pattern must not generate a new winning class. On the contrary, the same F_2 unit should win, with a higher activation than all the other recognition units.

This implies that the norm of the initial weight-vector has to be smaller than the one it has after several training cycles. If J $(1 \leq J \leq M)$ is the actual winning unit in F_2, then equation 9.6 is given by the theory:

$$||\mathbf{z}^J|| \rightarrow \left|\left|\frac{\mathbf{u}}{1-d}\right|\right| = \frac{1}{1-d}, \tag{9.6}$$

where $\mathbf{z}^J$ is the the weight vector of the links from the F_1 units to the Jth F_2 unit and where d is a parameter, described below.

If all initial values $z_{ij}(0)$ are presumed to be equal, this means:

$$z_{ij}(0) \leq \frac{1}{(1-d)\sqrt{N}} \qquad \forall\, 1 \leq i \leq N, 1 \leq j \leq M. \tag{9.7}$$

If equality is chosen in equation 9.7, then ART2 will be as sensitive as possible.

To transform the inequality 9.7 to an equation, in order to compute values, we introduce another parameter γ and get:

$$z_{ij}(0) = \frac{1}{\gamma(1-d)\sqrt{N}} \qquad \forall\, 1 \leq i \leq N, 1 \leq j \leq M, \tag{9.8}$$

where $\gamma \geq 1$.

To initialize an ART2 network, the function `ART2_Weights` has to be selected. Specify the parameters d and γ as the first and second initialization parameter. (A description of parameter d is given in the subsection on the ART2 learning function.) Finally press the INIT -button to initialize the net.

WARNING! You should always use `ART2_Weights` to initialize ART2 networks. When using another SNNS initialization function the behavior of the simulator during learning is not predictable, because not only the trainable links will be initialized, but also the fixed weights of the network.

ART2 Learning Function For the ART2 learning function `ART2` there are various parameters to specify. Here is a list of all parameters known from the theory:

- ρ Vigilance parameter. (first parameter of the learning and update function). ρ is defined on the interval $0 \leq \rho \leq 1$. For some reason, described in [Her92] only the following interval makes sense: $\frac{1}{2}\sqrt{2} \leq \rho \leq 1$.
- a Strength of the influence of the lower level in F_1 by the middle level. (second parameter of the learning and update function). Parameter a defines the importance of the expection of F_2, propagated to F_1: $a > 0$. Normally a value of $a \gg 1$ is chosen to assure quick stabilization in F_1.
- b Strength of the influence of the middle level in F_1 by the upper level. (third parameter of the learning and update function). For parameter b things are similar to parameter a. A high value for b is even more important, because otherwise the network could become instable ([CG87b]). $b > 0$, normally:$b \gg 1$.
- c Part of the length of vector $\mathbf{p}$ (units $\mathbf{p}_1 \ldots \mathbf{p}_N$) used to compute the error. (fourth parameter of the learning and update function). Choose c within $0 < c < 1$.

d Output value of the F_2 winner unit. You won't have to pass d to `ART2`, because this parameter is already needed for initialization. So you have to enter the value, when initializing the network (see subsection on the initialization function). Choose d within $0 < d < 1$. The parameters c and d are dependent on each other. For reasons of quick stabilization c should be chosen as follows: $0 < c \ll 1$. On the other hand c and d have to fit the following condition: $0 \ll \frac{cd}{1-d} \leq 1$.

e Prevents from division by zero. Since this parameter does not help to solve essential problems, it is implemented as a fix value within the SNNS source code.

Θ Kind of threshold. For $0 \leq x, q \leq \Theta$ the activation values of the units $\mathbf{x}_i$ and $\mathbf{q}_i$ only have small influence (if any) on the middle level of F_1. The output function f of the units $\mathbf{x}_i$ and $\mathbf{q}_i$ takes Θ as its parameter. Since this noise function is continuously differentiable, it is called `Out_ART2_Noise_ ContDiff` in SNNS. Alternatively a piecewise linear output function may be used. In SNNS the name of this function is `Out_ART2_Noise_PLin`. Choose Θ within $0 \leq \Theta < 1$.

To train an ART2 network, make sure, you have chosen the learning function `ART2`. As a first step initialize the network with the initialization function `ART2_Weights` described above. Then set the five parameters ρ, a, b, c and Θ, in the parameter windows 1 to 5 in both the `LEARN` and `UPDATE` lines of the control panel. Example values are 0.9, 10.0, 10.0, 0.1, and 0.0. Then select the number of learning cycles, and finally use the buttons SINGLE and ALL to train a single pattern or all patterns at a time, respectively.

ART2 Update Functions Again two update functions for ART2 networks have been implemented:

- `ART2_Stable`
- `ART2_Synchronous`.

Meaning and usage of these functions are equal to their equivalents of the ART1 model. For both of them the parameters ρ, a, b, c and Θ have to be defined in the row of update parameters in the control panel.

9.12.3 ARTMAP

9.12.3.1 Structure of an ARTMAP Network

Since an ARTMAP network is based on two networks of the ART1 model, it is useful to know how ART1 is realized in SNNS. Having taken two of the ART1 (ART^a and ART^b) networks as they were defined in section 9.12.1, we add several units that represent the MAP field. The connections between ART^a and the MAP field, ART^b and the MAP field, as well as those within the MAP field are shown in figure 9.14. The figure lacks the full connection from the F_2^a layer to the F^{ab} layer and those from each F_2^b unit to its respective F^{ab} unit and vice versa.

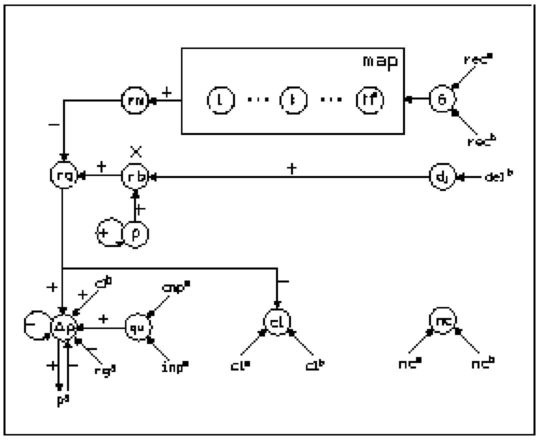

Figure 9.14: The MAP field with its control units.

The map field units represent the categories, onto which the ART^a classes are mapped[9]. The **G** unit is the MAP field gain unit. The units **rm** (reset map), **rb** (reset F_2^b), **rg** (reset general), ρ (vigilance) and $\mathbf{d}_1$ (delay 1) represent the inter-ART-reset control. $\Delta\rho$ and **qu** (quotient) have to realize the Match-Tracking-Mechanism and **cl** (classified) and **nc** (not classifiable) again show whether a pattern has been *classified* or was *not classifiable*.

9.12.3.2 Using ARTMAP Networks in SNNS

ARTMAP Initialization Function Since the trainable weights of an ARTMAP network are primarily the ones of the two ART1 networks ART^a and ART^b, it is easy to explain the ARTMAP initialization function `ARTMAP_Weights`. To use this function you have to select `ARTMAP_Weights` from the menu of the initialization functions. For `ARTMAP_Weights` you have to set the four parameters β^a, γ^a, β^b and γ^b. You can look up the meaning of each pair $\beta^?$, $\gamma^?$ in section 9.12.1.2, for the respective ART$^?$-part of the network.

ARTMAP Learning Function Select the ARTMAP learning function `ARTMAP` from the menu of the learning functions. Specify the three parameters $\overline{\rho^a}$, ρ^b and ρin the `LEARN` row of the control panel. Example values could be 0.7, 1.0, 1.0, and 0.0. $\overline{\rho^a}$ is the initial vigilance parameter for the ART^a-part of the net, which may be modified by the Match-Tracking-Mechanism. ρ^b is the vigilance parameter for the ART^b-part and ρ is the one for the Inter-ART-Reset control.

ARTMAP Update Functions For ARTMAP two update functions have been implemented, as well:

- `ARTMAP_Stable`

[9]Different ART^a classes may be mapped onto the same category.

- `ARTMAP_Synchronous`.

`ARTMAP_Stable` is again used to propagate a pattern through the network until a stable state is reached, while `ARTMAP_Synchronous` does only perform one propagation step at a time. For both of the functions the parameters $\overline{\rho^a}$, ρ^b and ρ have to be specified in the line for update parameters of the control panel. The usage is the same as it is for ART1 and ART2 networks.

9.12.4 Topology of ART Networks in SNNS

The following tables are an exact description of the topology requirements for the ART models ART1, ART2 and ARTMAP. For ARTMAP the topologies of the two ART1-parts of the net are the same as the one shown in the ART1 table.

ART2

unit definition					**connections**	
unit name	top. type	activation function	output function	site names	target unit	target site
$\mathbf{inp}_i$	i	`Act_Identity`	`Out_Identity`		$\mathbf{w}_i$ $\mathbf{r}_i$	
$\mathbf{w}_i$	h	`Act_ART2_Identity`	`Out_Identity`		$\mathbf{x}_i$	
$\mathbf{x}_i$	h	`Act_ART2_NormW`	*signal function*[1]		$\mathbf{v}_i$	
$\mathbf{v}_i$	h	`Act_ART2_Identity`	`Out_Identity`		$\mathbf{u}_i$	
$\mathbf{u}_i$	h	`Act_ART2_NormV`	`Out_Identity`		$\mathbf{p}_i$ $\mathbf{w}_i$	
$\mathbf{p}_i$	h	`Act_ART2_Identity`	`Out_Identity`		$\mathbf{q}_i$ $\mathbf{r}_i$ $\mathbf{rec}_j\ \forall j$	
$\mathbf{q}_i$	h	`Act_ART2_NormP`	*signal function*[1]		$\mathbf{v}_i$	
$\mathbf{r}_i$	h	`Act_ART2_NormIP`	`Out_Identity`			
$\mathbf{rec}_j$	s	`Act_ART2_Rec`	`Out_Identity`		$\mathbf{p}_i\ \forall i$ $\mathbf{rst}_j$	
$\mathbf{rst}_j$	h	`Act_ART2_Rst`	`Out_Identity`		$\mathbf{rec}_j$	

[1] either `Out_ART2_Noise_ContDiff` or `Out_ART2_Noise_PLin`.

ARTMAP

site definition		
site name	site function	
ARTa_G	Site_at_least_1	
ARTb_G	Site_at_least_1	
ARTb_rb	Site_WeightedSum	
rho_rb	Site_WeightedSum	
npa_qu	Site_Reciprocal	
cmpa_qu	Site_WeightedSum	

unit definition					connections	
unit name	top. type	activation function	output function	site names	target unit	target site
$\mathbf{map}_j$	h	Act_at_least_2	Out_Identity		**rm**	
cl	h	Act_at_least_2	Out_Identity			
nc	h	Act_at_least_1	Out_Identity			
G	h	Act_exactly_1	Out_Identity	ARTa_G ARTb_G	$\mathbf{map}_j\ \forall j$	
$\mathbf{d}_1$	h	Act_Identity	Out_Identity		**rb**	ARTb_rb
rb	h	Act_Product	Out_Identity	ARTb_rb rho_rb	**rg**	
rm	h	Act_Identity	Out_Identity		**rg**	
rg	h	Act_less_than_0	Out_Identity		**drho** **cl**	
rho	h	Act_Identity	Out_Identity		**rho** **rb**	 rho_rb
qu	h	Act_Product	Out_Identity	inpa_qu cmpa_qu	**drho**	
drho	h	Act_ARTMAP_DRho	Out_Identity		**drho** $\mathbf{rho}^a$	
$\mathbf{inp}_i^a$	see ART1 table				**qu**	inpa_qu
$\mathbf{cmp}_i^a$	see ART1 table				**qu**	cmpa_qu
$\mathbf{rec}_i^a$	see ART1 table				**G**	ARTa_G
$\mathbf{rec}_i^b$	see ART1 table				**G**	ARTb_G
$\mathbf{del}_i^b$	see ART1 table				$\mathbf{d}_1$	
$\mathbf{cl}^a$	see ART1 table				**cl**	
$\mathbf{cl}^b$	see ART1 table				**cl** **drho**	
$\mathbf{nc}^a$	see ART1 table				**nc**	
$\mathbf{nc}^b$	see ART1 table				**nc**	
$\mathbf{rg}^a$	see ART1 table				**drho**	
$\mathbf{rho}^a$	see ART1 table				**drho**	

ART1 and ART1-parts of ARTMAP (ART^a, ART^b)

site definition	
site name	site function
rst_self	Site_WeightedSum
rst_signal	Site_at_least_2
inp_g1	Site_at_least_1
rec_g1	Site_at_most_0
inp_ri	Site_WeightedSum
rho_ri	Site_WeightedSum

unit definition					connections	
unit name	top. type	activation function	output function	site-names	target unit	target site
inp$_i$	i	Act_Identity	Out_Identity		**cmp**$_i$ **g**$_1$ **ri** **g**$_2$	 inp_g1 inp_ri
cmp$_i$	h	Act_at_least_2	Out_Identity		**rec**$_j$ $\forall j$ **rc**	
rec$_j$	s	Act_Identity	Out_Identity		**del**$_j$ **g**$_1$	 rec_g1
del$_j$	h	Act_at_least_2	Out_Identity		**cmp**$_i$ $\forall i$ **d**$_1$ **rst**$_j$	 rst_signal
d$_1$	h	Act_at_least_1	Out_Identity		**d**$_2$	
d$_2$	h	Act_at_least_1	Out_Identity		**d**$_3$	
d$_3$	h	Act_at_least_1	Out_Identity		**cl**	
rst$_j$	h	Act_at_least_1	Out_Identity	rst_self rst_signal	**rst**$_j$ **rec**$_j$	rst_self
cl	h	Act_at_least_1	Out_Identity			
nc	h	Act_ART1_NC for ARTMAP: Act_ARTMAP_NCa, Act_ARTMAP_NCb	Out_Identity			
g$_1$	h	Act_at_least_2	Out_Identity	inp_g1 rec_g1	**cmp**$_i$ $\forall i$	
g$_2$	h	Act_at_most_0	Out_Identity		**rec**$_j$ $\forall j$ **cl**	
ri	h	Act_Product	Out_Identity	inp_ri rho_ri	**rg**	
rc	h	Act_Identity	Out_Identity		**rg**	
rg	h	Act_less_than_0	Out_Identity		**rec**$_j$ $\forall j$ **rst**$_j$ $\forall j$ **cl**	 rst_signal
rho	h	Act_Identity	Out_Identity		**rho** **ri**	 rho_ri

9.13 Self-Organizing Maps (SOMs)

9.13.1 SOM Fundamentals

The Self-Organizing Map (SOM) algorithm of Kohonen, also called Kohonen feature map, is one of the best known artificial neural network algorithms. In contrast to most other algorithms in SNNS, it is based on unsupervised learning. SOMs are a unique class of neural networks, since they construct topology-preserving mappings of the training data where the location of a unit carries semantic information. Therefore, the main application of this algorithm is clustering of data, obtaining a two-dimensional display of the input space that is easy to visualize.

Self-Organizing Maps consist of two layers of units: A one dimensional input layer and a two dimensional competitive layer, organized as a 2D grid of units. This layer can neither be called hidden nor output layer, although the units in this layer are listed as hidden units within SNNS. Each unit in the competitive layer holds a weight (reference) vector, W_i, that, after training, resembles a different input pattern. The learning algorithm for the SOM accomplishes two important things:

1. clustering the input data
2. spatial ordering of the map so that similar input patterns tend to produce a response in units that are close to each other in the grid.

Before starting the learning process, it is important to initialize the competitive layer with normalized vectors. The input pattern vectors are presented to all competitive units in parallel and the best matching (nearest) unit is chosen as the winner. Since the vectors are normalized, the similarity between the normalized input vector X = (x_i) and the reference units $W_j = (w_{ij})$ can be calculated using the dot product:

$$Net_j(t) = X \cdot W_j = \sum_{i=1}^{n} x_i(t) w_{ij}(t)$$

The vector[10] W_c most similar to X is the one with the largest dot product with X:

$$Net_c(t) = \max_j \{Net_j(t)\} = X \cdot W_c$$

The topological ordering is achieved by using a spatial neighborhood relation between the competitive units during learning. I.e. not only the best-matching vector, with weight W_c, but also its neighborhood [11] N_c, is adapted, in contrast to a basic competitive learning algorithm like LVQ:

[10] c will be used as index for the winning unit in the competitive layer throughout this text

[11] Neighborhood is defined as the set of units within a certain radius of the winner. So $N(1)$ would be the the eight direct neighbors in the 2D grid; $N(2)$ would be $N(1)$ plus the 16 next closest; etc.

$$\begin{aligned}\Delta w_{ij}(t) &= e_j(t) \cdot (x_i(t) - w_{ij}(t)) & \text{for } j \in N_c \\ \Delta w_{ij}(t) &= 0 & \text{for } j \notin N_c\end{aligned}$$

where

$e_j(t) = h(t) \cdot e^{-(d_j/r(t))^2}$ (Gaussian Function)
d_j: distance between W_j and winner W_c
$h(t)$: adaptation height at time t with $0 \leq h(t) \leq 1$
$r(t)$: radius of the spatial neighborhood N_c at time t

The adaption height and radius are usually decreased over time to enforce the clustering process.

See [Koh88] for a more detailed description of SOMs and their theory.

9.13.2 SOM Implementation in SNNS

SNNS originally was not designed to handle units whose location carry semantic information. Therefore some points have to be taken care of when dealing with SOMs. For learning it is necessary to pass the horizontal size of the competitive layer to the learning function, since the internal representation of a map is different from its appearance in the display. Furthermore it is not recommended to use the graphical network editor to create or modify any feature maps. The creation of new feature maps should be carried out with the **BIGNET (Kohonen)** creation tool (see chapter 7).

9.13.2.1 The KOHONEN Learning Function

SOM training in SNNS can be performed with the learning function `Kohonen`. It can be selected from the list of learning functions in the control panel. Five parameters have to be passed to this learning function:

- **Adaptation Height (Learning Height)** The initial adaptation height $h(0)$ can vary between 0 and 1. It determines the overall adaptation strength.
- **Adaptation Radius (Learning Radius)** The initial adaptation radius $r(0)$ is the radius of the neighborhood of the winning unit. All units within this radius are adapted. Values should range between 1 and the size of the map.
- **Decrease Factor mult_H** The adaptation height decreases monotonically after the presentation of every learning pattern. This decrease is controlled by the decrease factor mult_H: $h(t+1) := h(t) \cdot mult\_H$
- **Decrease Factor mult_R** The adaptation radius also decreases monotonically after the presentation of every learning pattern. This second decrease is controlled by the decrease factor mult_R: $r(t+1) := r(t) * mult\_R$

- **Horizontal size** Since the internal representation of a network doesn't allow to determine the 2-dimensional layout of the grid, the horizontal size in units must be provided for the learning function. It is the same value as used for the creation of the network.

Note: After each completed training the parameters adaption height and adaption radius are updated in the control panel to reflect their new values. So when training is started anew, it resumes at the point where it was stopped last. Both mult_H and mult_R should be in the range $(0, 1]$. A value of 1 consequently keeps the adaption values at a constant level.

9.13.2.2 The Kohonen Update Function

A special update function (`Kohonen_Order`) is also provided in SNNS. This function has to be used since it is the only one that takes care of the special ordering of units in the competitive layer. If another update is selected an "Error: Dead units in the network" may occur when propagating patterns.

9.13.2.3 The Kohonen Init Function

Before a SOM can be trained, its weights have to be initialized using the init function `Kohonen_Weights`. This init function first initializes all weights with random values between the specified borders. Then it normalizes all links on a per-unit basis. Some of the internal values needed for later training are also set here. It uses the same parameters as `CPN_Weights` (see section 9.5.2).

9.13.2.4 The Kohonen Activation Functions

The Kohonen learning algorithm does not use the activation functions of the units. Therefore it is basically unimportant which activation function is used in the SOM. For display and evaluation reasons, however, the two activation functions `Act_Component` and `Act_Euclid` have been implemented in SNNS. `Act_Euclid` copies the Euclidian distance of the unit from the training pattern to the unit activation, while `Act_Component` writes the weight to one specific component unit into the activation of the unit.

9.13.2.5 Building and Training Self-Organizing Maps

Since any modification of a Self-Organizing Map in the 2D display like the creation, deletion or movement of units or weights may destroy the relative position of the units in the map we strongly recommend to generate these networks only with the available **BIGNET (Kohonen)** tool. See also chapter 7 for detailed information on how to create networks. Outside xgui you can also use the tool **convert2snns**. Information on this program can be found in the respective README file in the directory SNNSv3.3/tools/doc. Note:

Any modification of the units after the creation of the network may result in undesired behavior!

To train a new feature map with SNNS, set the appropriate standard functions: select init function **KOHONEN_Weights**, update function **Kohonen_Order** and learning function **Kohonen**. Remember: There is no special activation function for Kohonen learning, since setting an activation function for the units doesn't affect the learning procedure. To visualize the results of the training, however, one of the two activation functions `Act_Euclid` and `Act_Componnent` has to be selected. For their semantics see section 9.13.2.6.

After providing patterns (ideally normalized) and assigning reasonable values to the learning function, the learning process can be started. To get a proper appearance of SOMs in the 2D-display set the **grid width** to 16 and turn off the unit labeling and link display in the display panel.

When a learning run is completed the adaption height and adaption radius parameters are automatically updated in the control panel to reflect the actual values in the kernel.

9.13.2.6 Evaluation Tools for SOMs

When the results of the learning process are to be analyzed, the tools described here can be used to evaluate the qualitative properties of the SOM. In order to provide this functionality, a special panel was added. It can be called from the manager panel by clicking the `KOHONEN` button and is displayed in figure 9.15. Yet, the panel can only be used in combination with the control panel.

Figure 9.15: The additional KOHONEN (control) panel

1. **Euclidian distance**
   The distance between an input vector and the weight vectors can be visualized using a distance map. This function allows using the SOM as a classifier for arbitrary input patterns: Choose **Act_Euclid** as activation function for the hidden units, then use the TEST button in the control panel to see the distance maps of consecutive patterns. As green squares (big filled squares on B/W terminals) indicate high activations, green squares here mean big distances, while blue squares represent small distances. Note: The input vector is not normalized before calculating the distance to the competitive units. This doesn't affect the qualitative appearance of the distance maps, but offers the advantage of evaluating SOMs that were generated by different SOM-algorithms (learning without normalization). If the dot product as similarity measure is to be used select **Act_Identity** as activation function for the hidden units.

2. **Component maps**
   To determine the quality of the clustering for each component of the input vector use this function of the SOM analyzing tool. Due to the topology-preserving nature of the SOM algorithm, the component maps can be compared after printing, thereby detecting correlations between some components: Choose the activation function **Act_Component** for the hidden units. Just like displaying a pattern, component maps can be displayed using the **LAYER** buttons in the **KOHONEN** panel. Again, green squares represent large, positive weights.

3. **Winning Units**
   The set of units that came out as winners in the learning process can also be displayed in SNNS. This shows the distribution of patterns on the SOM. To proceed, turn on **units top** in the setup window of the display and select the **winner** item to be shown. New winning units will be displayed without deleting the existing, which enables tracing the temporal development of clusters while learning is in progress. The display of the winning units is refreshed by pressing the WINNER button again.

   Note: Since the winner algorithm is part of the KOHONEN learning function, the learning parameters must be set as if learning is to be performed.

9.14 Autoassociative Networks

9.14.1 General Characteristics

Autoassociative networks store single instances of items, and can be thought of in a way similar to human memory. In an autoassociative network each pattern presented to the network serves as both the input and the output pattern. Autoassociative networks are typically used for tasks involving pattern completion. During retrieval, a probe pattern is presented to the network causing the network to display a composite of its learned patterns most consistent with the new information.

Autoassociative networks typically consist of a single layer of nodes with each node representing some feature of the environment. However in SNNS they are represented by two layers to make it easier to compare the input to the output. The following section explains the layout in more detail[12].

Autoassociative networks must use the update function `RM_Synchronous` and the initialization function `RM_Random_Weights`. The use of others may destroy essential characteristics of the autoassociative network. Please note, that the update function `RM_Synchronous` needs as a parameter the number of iterations performed before the network output is computed. 50 has shown to be very suitable here.

All the implementations of autoassociative networks in SNNS report error as the sum of squared error between the input pattern on the world layer and the resultant pattern on

[12] For any comments or questions concerning the implementation of an autoassociative memory please refer to Jamie DeCoster at jamie@psych.purdue.edu

the learning layer after the pattern has been propagated a user defined number of times.

9.14.2 Layout of Autoassociative Networks

An autoassociative network in SNNS consists of two layers: A layer of **world** units and a layer of **learning** units. The representation on the world units indicates the information coming into the network from the outside world. The representation on the learning units represents the network's current interpretation of the incoming information. This interpretation is determined partially by the input and partially by the network's prior learning.

Figure 9.16 shows a simple example network. Each unit in the world layer sends input to exactly one unit in the learning layer. The connected pair of units corresponds to a single node in the typical representation of autoassociative networks. The link from the world unit always has a weight of 1.0, and is unidirectional from the world unit to the learning unit. The learning units are fully interconnected with each other.

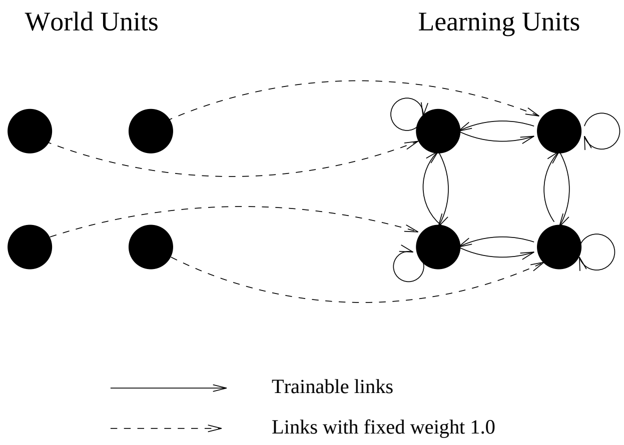

Figure 9.16: A simple Autoassociative Memory Network

The links between the learning units change according to the selected learning rule to fit the representation on the world units. The links between the world units and their corresponding learning units are not affected by learning.

9.14.3 Hebbian Learning

In Hebbian learning weights between learning nodes are adjusted so that each weight better represents the relationship between the nodes. Nodes which tend to be positive or negative at the same time will have strong positive weights while those which tend to be opposite will have strong negative weights. Nodes which are uncorrelated will have weights near zero.

The general formula for Hebbian learning is

$$\Delta w_{ij} = n * input_i * input_j$$

where:

n	is the learning rate
$input_i$	is the external input to node i
$input_j$	is the external input to node j

9.14.4 McClelland & Rumelhart's Delta Rule

This rule is presented in detail in chapter 17 of [RM86]. In general the delta rule outperforms the Hebbian learning rule. The delta rule is also less likely so produce explosive growth in the network. For each learning cycle the pattern is propagated through the network ncycles (a learning parameter) times after which learning occurs. Weights are updated according to the following rule:

$$\Delta w_{ij} = n * d_i * a_j$$

where:

n	is the learning rate
a_j	is the activation of the source node
d_i	is the error in the destination node. This error is defined as the external input - the internal input.

In their original work McClelland and Rumelhart used an unusual activation function:

```
for unit i,
    if neti > 0
       delta ai = E * neti * (1 - ai) - D * ai
    else
       delta ai = E * neti * (ai + 1) - D * ai
```

where:

$neti$	is the net input to i (external + internal)
E	is the excitation parameter (here set to 0.15)
D	is the decay parameter (here set to 0.15)

This function is included in SNNS as `ACT_RM`. Other activation functions may be used in its place.

9.15 Partial Recurrent Networks

9.15.1 Models of Partial Recurrent Networks

9.15.1.1 Jordan Networks

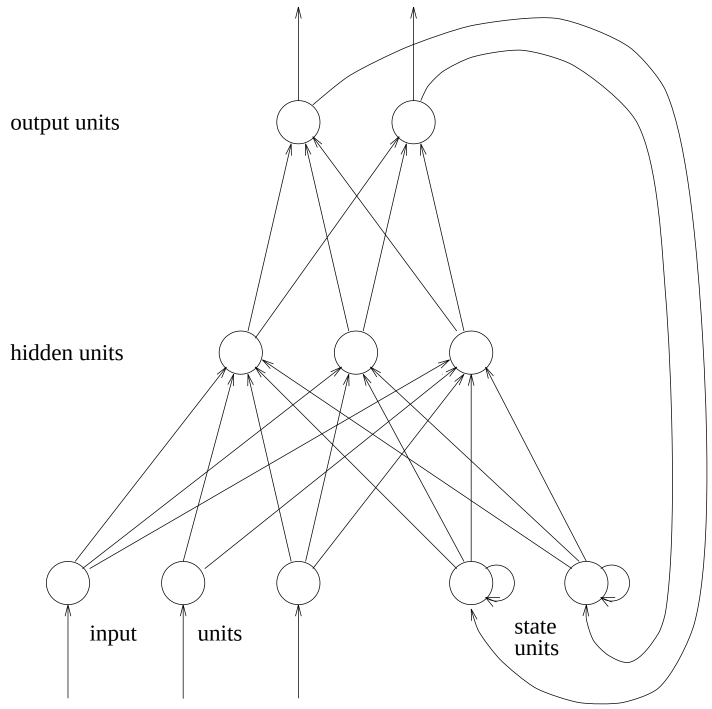

Figure 9.17: Jordan network

Literature: [Jor86b], [Jor86a]

9.15.1.2 Elman Networks

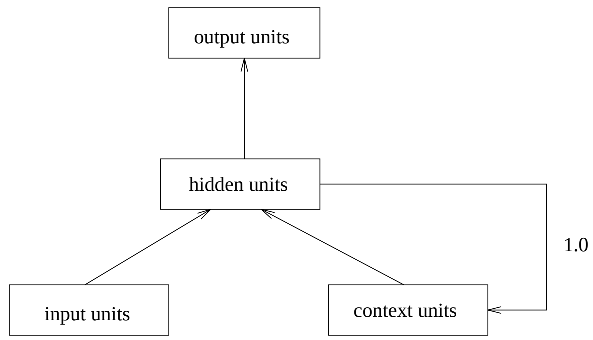

Figure 9.18: Elman network

Literature: [Elm90]

9.15.1.3 Extended Hierarchical Elman Networks

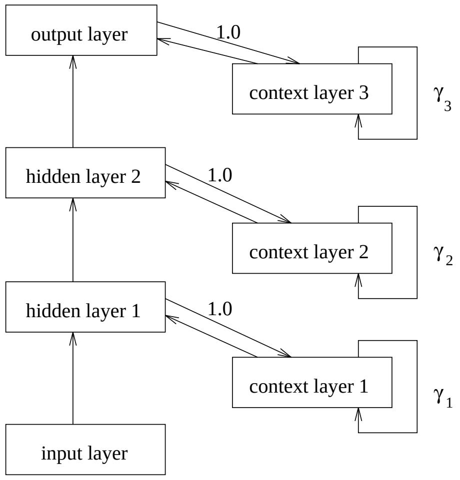

Figure 9.19: Extended Elman architecture

9.15.2 Working with Partial Recurrent Networks

In this subsection, the initialization, learning, and update functions for partial recurrent networks are described. These functions can not be applied to only the three network models described in the previous subsection. They can be applied to a broader class of partial recurrent networks. Every partial recurrent network, that has the following restrictions, can be used:

- If after the deletion of all context units and the links to and from them, the remaining network is a simple feedforward architecture with no cycles.
- Input units must not get input from other units.
- Output units may only have outgoing connections to context units, but not to other units.
- Every unit, except the input units, has to have at least one incoming link. For a context unit this restriction is already fulfilled when there exists only a self recurrent link. In this case the context unit receives its input only from itself.

In such networks all links leading to context units are considered as recurrent links.

Thereby the user has a lot of possibilities to experiment with a great variety of partial recurrent networks. E.g. it is allowed to connect context units with other context units.

Note: context units are realized as special hidden units. All units of type special hidden are assumed to be context units and are treated like this.

9.15.2.1 The Initialization Function JE_Weights

The initialization function `JE_Weights` requires the specification of five parameters:

- α, β : The weights of the forward connections are randomly chosen from the interval $[\alpha\,;\beta]$.
- λ : Weights of self recurrent links from context units to themselves. Simple Elman networks use $\lambda = 0$.
- γ : Weights of other recurrent links to context units. This value is often set to 1.0.
- ψ : Initial activation of all context units.

These values are to be set in the `INIT` line of the control panel in the order given above.

9.15.2.2 Learning Functions

By deleting all recurrent links in a partial recurrent network, a simple feedforward network remains. The context units have now the function of input units, i.e. the total network input consists of two components. The first component is the pattern vector, which was the only input to the partial recurrent network. The second component is a state vector. This state vector is given through the next–state function in every step. By this way the behavior of a partial recurrent network can be simulated with a simple feedforward network, that receives the state not implicitly through recurrent links, but as an explicit part of the input vector. In this sense, backpropagation algorithms can easily be modified for the training of partial recurrent networks in the following way:

1. Initialization of the context units. In the following steps, all recurrent links are assumed to be not existent, except in step 2(f).
2. Execute for each pattern of the training sequence the following steps:
   - input of the pattern and forward propagation through the network
   - calculation of the error signals of output units by comparing the computed output and the teaching output
   - back propagation of the error signals
   - calculation of the weight changes
   - only on–line training: weight adaption
   - calculation of the new state of the context units according to the incoming links
3. Only off–line training: weight adaption

In this manner, the following learning functions have been adapted for the training of partial recurrent networks like Jordan and Elman networks:

- `JE_BP`: Standard Backpropagation for partial recurrent networks

- `JE_BPMomentum`: Standard Backpropagation with Momentum–Term for partial recurrent networks
- `JE_Quickprop`: Quickprop for partial recurrent networks
- `JE_Rprop`: Rprop for partial recurrent networks

The parameters for these learning functions are the same as for the regular feedforward versions of these algorithms (see section 4.3).

For training a network with one of these functions a method called teacher forcing is used. That means that during the training phase the output units propagate the teaching output instead of their produced output to successor units (if there are any).

9.15.2.3 Update Functions

Two new update functions have been implemented for partial recurrent networks:

- `JE_Order`:
  This update function propagates a pattern from the input layer to the first hidden layer, then to the second hidden layer, etc. and finally to the output layer. After this follows a synchronous update of all context units.
- `JE_Special`:
  This update function can be used for iterated long-term predictions. If the actual prediction value p_i is part of the next input pattern inp_{i+1} to predict the next value p_{i+1}, the input pattern inp_{i+1} can not be generated before the needed prediction value p_i is available. For long-term predictions many input patterns have to be generated in this manner. To generate these patterns manually means a lot of effort.

  Using the update function `JE_Special` these input patterns will be generated dynamically. Let n be the number of input units and m the number of output units of the network. `JE_Special` generates the new input vector with the output of the last $n - m$ input units and the outputs of the m output units. The usage of this update function requires $n > m$. The propagation of the new generated pattern is done like using `JE_Update`. The number of the actual pattern in the control panel has no meaning for the input pattern when using `JE_Special`.

9.16 Stochastic Learning Functions

The monte carlo method and simulated annealing are widely used algorithms for solving any kind of optimization problems. These stochastic functions have some advantages over other learning functions. They allow any net structure, any type of neurons and any type of error function. Even every neuron of a net could have another learning function and any number of links.

9.16.1 Monte-Carlo

Monte-Carlo learning is an easy way to determine weights and biases of a net. At every learning cycle all weights and biases are chosen by random in the Range $[Min, Max]$. Then the error is calculated as summed squared error of all patterns. If the error is lower than the previous best error, the weights and biases are stored. This method is not very efficient but useful for finding a good start point for another learning algorithm.

9.16.2 Simulated Annealing

Simulated annealing is a more sophisticated method for finding the global minima of a error surface. In contrast to monte carlo learning only one weight or bias is changed at a learning cycle. Dependant on the error development and a system temperature this change is accepted or rejected. One of the advantages of simulated annealing is that learning does not get stuck in local minima.

At the beginning of learning the temperature T is set to T_0. Each training cycle consists of the following four steps.

1. Change one weight or bias by random in the range $[Min, Max]$.
2. Calculate the net error as sum of the given error function for all patterns.
3. Accept change if the error decreased or if the error increased by ΔE with the probability p given by: $p = \exp\left(\frac{-\Delta E}{T}\right)$
4. Decrease the temperature: $T = T \cdot deg$

The three implemented simulated annealing functions only differ in the way the net error is calculated. Sim_Ann_SS calculates a summed squared error like the backpropagation learning functions; Sim_Ann_WTA calculates a winner takes all error; and Sim_Ann_WWTA calculates a winner takes all error and adds a term corresponding to the security of the winner takes all decision.

Chapter 10

Pruning Algorithms

This chapter describes the four pruning functions which are available in SNNS. The first section of this chapter introduces the common ideas of pruning functions, the second takes a closer look at the theory of the implemented algorithms and the last part gives guidance for the use of the methods. Detailed description can be found in [Bie94] (for "non-contributing units") and [Sch94] (for the rest).

10.1 Background of Pruning Algorithms

Pruning algorithms try to make neural networks smaller by pruning unnecessary links or units, for different reasons:

- It is possible to find a fitting architecture this way.
- The cost of a net can be reduced (think of runtime, memory and cost for hardware implementation).
- The generalization[1] can (but need not) be improved.
- Unnecessary input units can be pruned in order to give evidence of the relevance of input values.

Pruning algorithms can be rated according to two criterions:

- *What* will be pruned? We distinguish weight pruning and node pruning. Special types of node pruning are input pruning and hidden unit pruning.
- *How* will be pruned? The most common possibilities are *penalty term algorithms* (like Backpropagation with Weight Decay, see section 9.1.4) and *sensitivity algorithms* which are described in this chapter.

Sensitivity algorithms perform training and pruning of a neural net alternately, according to the algorithm in figure 10.1.

[1] Generalization: ability of a neural net to recognize unseen patterns (test set) after training

1. Choose a reasonable network architecture.
2. Train the net with backpropagation or any similar learning function into a minimum of the network.
3. Compute the *saliency* (relevance for the performance of the network) of each element (link or unit respectively).
4. Prune the element with the smallest saliency.
5. Retrain the net (into a minimum again).
6. If the net error is not too big, repeat the procedure from step 3 on.
7. (optional) Recreate the last pruned element in order to achieve a small net error again.

Figure 10.1: Algorithm for sensitivity algorithms

10.2 Theory of the implemented algorithms

There are different approaches to determine the saliency of an element in the net. This section introduces the implemented sensitivity pruning algorithms.

10.2.1 Magnitude Based Pruning

This is the simplest weight pruning algorithm. After each training, the link with the smallest weight is removed. Thus the saliency of a link is just the absolute size of its weight. Though this method is very simple, it rarely yields worse results than the more sophisticated algorithms.

10.2.2 Optimal Brain Damage

Optimal Brain Damage (OBD) approximates the change of the error function when pruning a certain weight. A Taylor series is used for the approximation:

$$\begin{aligned} \delta E &= E(W + \delta W) - E(W) \\ &= \frac{\partial E(W)}{\partial W} \cdot \delta W + \frac{1}{2} \frac{\partial^2 E(W)}{\partial W^2} + R_2(W + \delta W) \end{aligned} \tag{10.1}$$

To simplify the computation, we assume that

- the net error function was driven into a minimum by training, so that the first term on the right side of equation (10.1) can be omitted;
- the net error function is locally quadratic, so that the last term in the equation can be left out;
- the remaining second derivative (*Hesse-matrix)* consists only of diagonal elements, which affects the second term in equation (10.1).

The result of all these simplifications reads as follows:

$$\delta E \;=\; \frac{1}{2} \sum_{1 \leq i,j \leq n_n} h_{(ij),(ij)} \cdot \delta w_{ij}^2 \tag{10.2}$$

Now it is necessary to compute the diagonal elements of the Hesse-Matrix. For the description of this and to obtain further information read [YLC90].

10.2.3 Optimal Brain Surgeon

Optimal Brain Surgeon (OBS, see [BH93]) was a further development of OBD. It computes the full Hesse-Matrix iteratively, which leads to a more exact approximation of the error function:

$$\delta E \;=\; \frac{1}{2}\, \delta W^T \cdot H \cdot \delta W \tag{10.3}$$

From equation (10.3), we form a minimization problem with the additional condition, that at least one weight must be set to zero:

$$\min_{(ij)} \left\{ \min_{\delta W} \left(\frac{1}{2}\, \delta W^T \cdot H \cdot \delta W \right) \mid E_{(ij)}^T \cdot \delta W \;+\; w_{ij} \;=\; 0 \right\} \tag{10.4}$$

and deduce a Lagrangian from that:

$$L \;=\; \frac{1}{2}\, \delta W^T \cdot H \cdot \delta W \;+\; \lambda \cdot (E_{(ij)}^T \cdot \delta W \;+\; w_{(ij)}) \tag{10.5}$$

where λ is an Lagrangian multiplier. This leads to

$$\delta W \;=\; -\frac{w_{ij}}{[H^{-1}]_{(ij),(ij)}} \cdot H^{-1} \cdot E_{(ij)} \tag{10.6}$$

$$L_{(ij)} \;=\; s_{ij} \;=\; \frac{1}{2} \frac{w_{(ij)}^2}{[H^{-1}]_{(ij),(ij)}} \tag{10.7}$$

Note that the weights of all links are updated.

The problem is, that the inverse of the Hesse-Matrix has to be computed to deduce saliency and weight change for every link. A sophisticated algorithm has been developed, but it is still very slow and takes much memory, so that you will get in trouble for bigger problems.

10.2.4 Skeletonization

Skeletonization ([MM89]) prunes units by estimating the change of the error function when the unit is removed (like OBS and OBD do for links). For each node, the *attentional strength* is introduced which leads to a different formula for the net input:

$$net_j \;=\; \sum_i w_{ij} \cdot \alpha_i \cdot o_i \tag{10.8}$$

Figure 10.2 illustrates the use of the attentional strength.

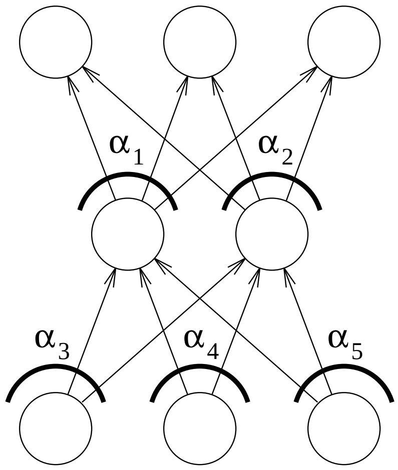

Figure 10.2: Neural network with attentional strength for each input and hidden neuron

Defining the relevance of a unit as the change in the error function while removing the unit we get

$$\rho_i \;=\; E_{\alpha_i=0} \;-\; E_{\alpha_i=1} \approx \left.\frac{\partial E}{\partial \alpha_i}\right|_{\alpha_i=1} \tag{10.9}$$

In order to compute the saliency, the linear error function is used:

$$E \;=\; E^l \;=\; \sum_j |t_{pj} - o_{pj}| \tag{10.10}$$

10.2.5 Non-contributing Units

This method uses statistical means to find units that don't contribute to the net's behavior. The net is subdivided into its layers, the output of each neuron is observed for the whole pattern set and units are removed that

- don't vary their output,
- always show the same output as another unit of the same layer or
- always show the opposite output of another unit of the same layer.

This function is the first part of the method introduced by [JS91]. For further information about the implementation read [Bie94].

10.3 Pruning Nets in SNNS

To use one of the pruning algorithms mentioned above, set the learning function in the control panel (options menu, see section 4.3) to `PruningFeedForward`.

Note: This information is saved with the net. Be sure to check the learning function each time you reload the net.

The pruning algorithms can be customized by changing the parameters in the pruning panel (see figure 10.3) which can be invoked by choosing `PRUNING` in the main menu. The figure shows the default values of the parameters.

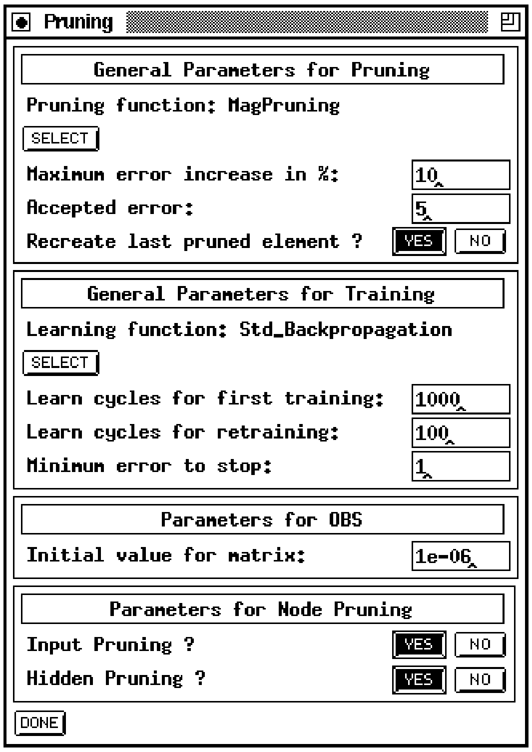

Figure 10.3: Pruning Panel

The first block of the panel controls the pruning, the second the embedded learning epochs and the last two specify parameters for certain pruning algorithms. To save the changes to the parameters and close the pruning panel, press the button `DONE`.

The current pruning function is shown in the box "`General Parameters for Pruning`". To select a different one, press the button `SELECT` which pops up a list of the available pruning functions, analogous to the change of the learning function.

There are two criterions to stop the pruning: The error after retraining must not exceed

- the error before the first pruning by more then a certain percentage determined by the user in the field "`Maximum error increase in %:`" and
- the absolute value given in the field "`Accepted error`"

Normally, the state of the net before the last (obviously too expensive) pruning is restored at the end. You can prevent this by switching the radio buttons next to "`Recreate last pruned element`" to `NO`.

The second box in the pruning panel ("`General Parameters for Learning`") allows to select the (subordinate) learning function in the same way as the pruning function. Only learning functions for feed-forward-nets appear in the list. You can select the number of epochs for the first training and each retraining separately. The training, however, stops when the absolute error falls short of the "`Minimum error to stop`". This prevents the net from overtraining.

The parameter for OBS in the third box is the initial value of the diagonal elements in the Hesse-Matrix. For the exact meaning of that parameter (to which OBS is said to be not very sensible) see [BH93].

The last box allows the user to choose which kind of neurons should be pruned by the node pruning methods. Input or hidden unit pruning can be selected by two sets of radio buttons.

Learning Parameters of the subordinate learning function have to be typed in in the control panel, as if the training would be processed by this function only. The field `CYCLES` in the control panel has no effect on pruning algorithms. To start pruning, press the button `DONE` in the pruning panel and the button `ALL` or `SINGLE` respectively in the control panel.

Chapter 11

3D-Visualization of Neural Networks

11.1 Overview of the 3D Network Visualization

This section presents a short overview over the 3D user interface. The following figures show the user interface with a simple three-layer network for the recognition of letters.

The info window is located in the upper left corner of the screen. There, the values of the units can be displayed and changed. Next to it, the 2D display is placed. This window is used to create and display the network topology. The big window below is used for messages from the kernel and the user interface. The control panel in the lower left corner controls the learning process. Above it, the 3D display is located which shows the 3D visualization of the network. The 3D control window, in the center of the screen, is used to control the 3D display.

In the upper part, the orientation of the network in space can be specified. The middle part is used for the selection of various display modes. In `SETUP` the basic settings can be selected. With `MODEL` the user can switch between solid and wire-frame model display. With `PROJECT` parallel or central projection can be chosen. `LIGHT` sets the illumination parameters, while `UNITS` lets the user select the values for visualizing the units. The display of links can be switched on with `LINKS`. `RESET` sets the network to its initial configuration. After a click to `FREEZE` the network is not updated anymore. The `DISPLAY` button opens the 3D-display window and `DONE` closes it again. In the lower part of the window, the z-coordinate for the network layers can be set.

11.2 Use of the 3D-Interface

11.2.1 Structure of the 3D-Interface

The 3D interface consists of three parts:

- the 2D $\rightarrow$ 3D transformation in the XGUI-display
- the 3D control panel
- the 3D display window

11.2.2 Calling and Leaving the 3D Interface

The 3D interface is called with the `GUI` button in the info panel. It opens the 3D Control panel which controls the network display. When the configuration file of a three dimensional network is loaded, the control panel and the display window are opened automatically, if this was specified in the configuration. No additional control panel may be opened if one is already open.

The 3D interface is left by pressing the `DONE` button in the control panel.

11.2.3 Creating a 3D-Network

11.2.3.1 Concepts

A three dimensional network is created with the network editor in the first 2D-display. It can be created in two dimensions as usual and then changed into 3D form by adding a z-coordinate. It may as well be created directly in three dimensions.

Great care was given to compatibility aspects on the extension of the network editor. Therefore a network is represented in exactly the same way as in the 2D case.

In the 2D representation each unit is assigned a unique (x, y) coordinate. The different layers of units lie next to each other. In the 3D representation these layers are to lie on top of each other. An additional z-coordinate may not simply be added, because this would lead to ambiguity in the 2D display.

Therefore an (x, y) offset by which all units of a layer are transposed against their position in the 2D display has to be computed for each layer. The distance of the layer in height corresponds to the z value. Only entire layers may be moved, i.e. all units of a layer have to be in the same z plane, meaning they must have the same z-coordinate. Figure 11.1 explains this behavior.

Therefore the network editor contains two new commands

Units 3d Z	: assigning a z-coordinate
Units 3d Move	: Moving a z-layer

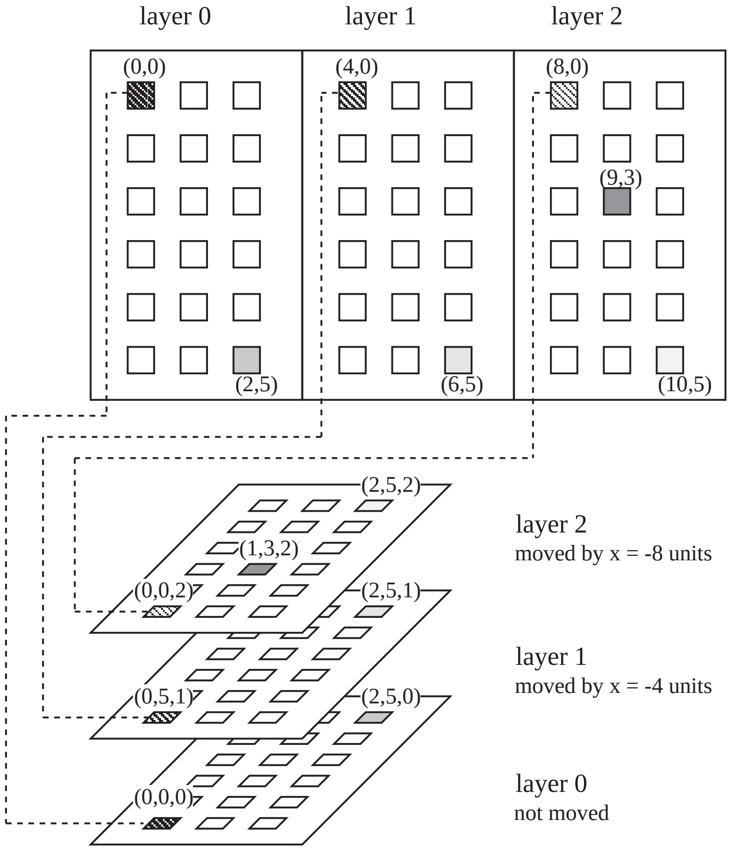

Figure 11.1: Layers in the 2D- and 3D-display

The event of 3D-creation is easily controlled by rotating the network in the 3D display by 90° to be able to see the network sideways. It may be useful to display the z-coordinates in the XGUI display (see 11.2.3.4).

The user is advised to create a 3D network first as a wire-frame model without links for much faster screen display.

11.2.3.2 Assigning a new z-Coordinate

The desired new z-coordinate may be entered in the setup panel of the 2D-display, or in the z-value panel of the 3D-control panel. The latter is more convenient, since this panel is always visible. Values between -32768 and +32767 are legal.

With the mouse all units are selected which are to receive the new z-coordinate.

With the key sequence `U 3 Z` (for **U**nits **3**d **Z**) the units are assigned the new value.

Afterwards all units are deselected.

11.2.3.3 Moving a z-Plane

From the plane to be moved, one unit is selected as a reference unit in the 2D display. Then the mouse is moved to the unit in the base layer above which the selected unit is to be located after the move.

With the key sequence `U 3 M` (for **U**nits **3**d **M**ove) all units of the layer are moved to the current z-plane.

The right mouse button deselects the reference unit.

11.2.3.4 Displaying the z-Coordinates

The z-values of the different units can be displayed in the 2D-display. To do this, the user activates the setup panel of the 2D-display with the button `SETUP`. The button `SHOW`, next to the entry `units top` opens a menu where `z-value` allows the display of the values.

The z-values may also be displayed in the 3D-display. For this, the user selects in the 3D-control panel the buttons `UNITS`, then `TOP LABEL` or `BOTTOM LABEL` and finally `Z-VALUE`. (see also chapter 11.2.4.6)

11.2.3.5 Example Dialogue to Create a 3D-Network

The following example is to demonstrate the rearranging of a normal 2D network for three dimensional display. As example network, the letter classifier LETTERS.NET is used.

In the 2D-display, the network looks like in figure 11.2:

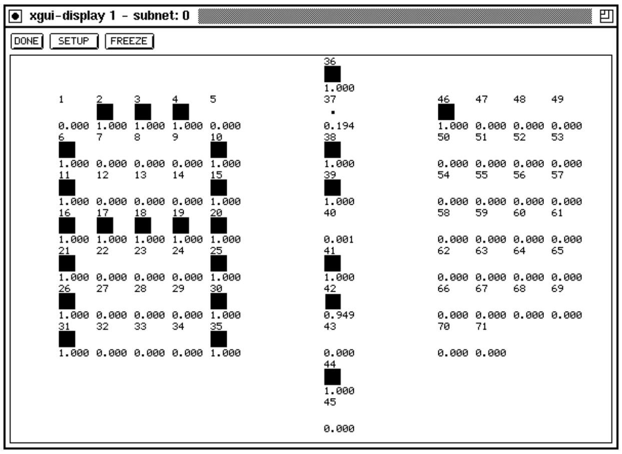

Figure 11.2: 2D-display

One scales the net with `scale` `-` in the transformation panel, then it looks like figure 11.3 (left). After a rotation with `rotate` `+` by 90° around the x-axis the network looks like figure 11.3 (right).

Now the middle layer is selected in the 2D-display (figure 11.4, left).

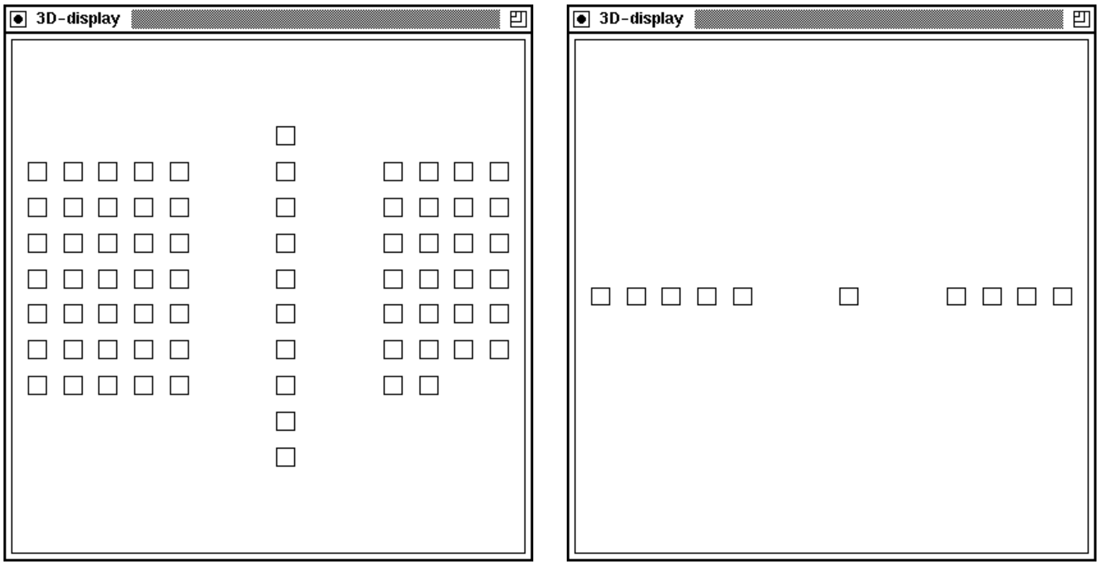

Figure 11.3: Scaled network (left) and network rotated by 90° (right)

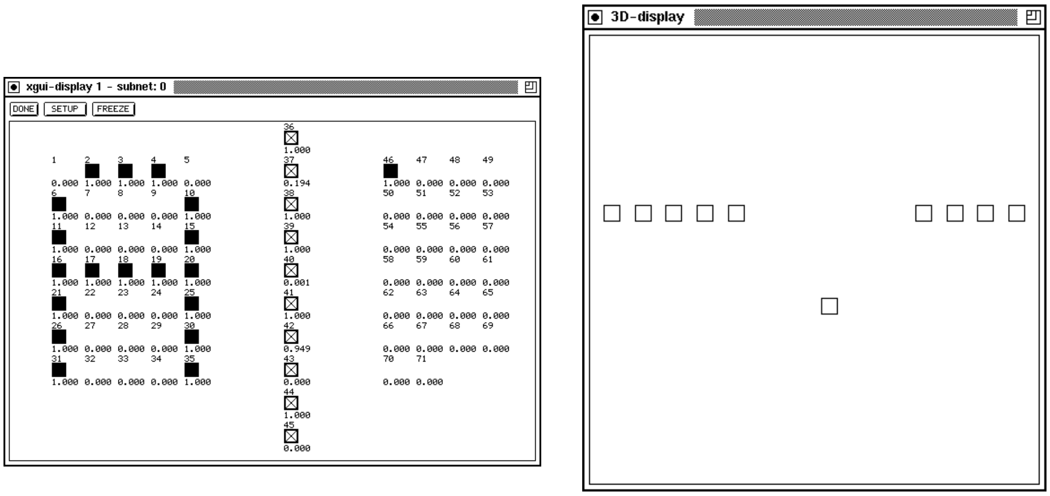

Figure 11.4: Selection of one layer (left) and assigning a z-value (right)

To assign the z-coordinate to the layer, the `z-value` entry in the 3D-control panel is set to three. Then one moves the mouse into the 2D-display and enters the key sequence "'`U 3 Z`"'. This is shown in figure 11.4 (right).

Figure 11.5: Selection of a reference unit (left) and moving a plane (right)

Now the reference unit must be selected (figure 11.5, left).

To move the units over the zero plane, the mouse is moved in the XGUI display to the position x=3, y=0 and the keys "'`U 3 M`"' are pressed. The result is displayed in figure 11.5 (right).

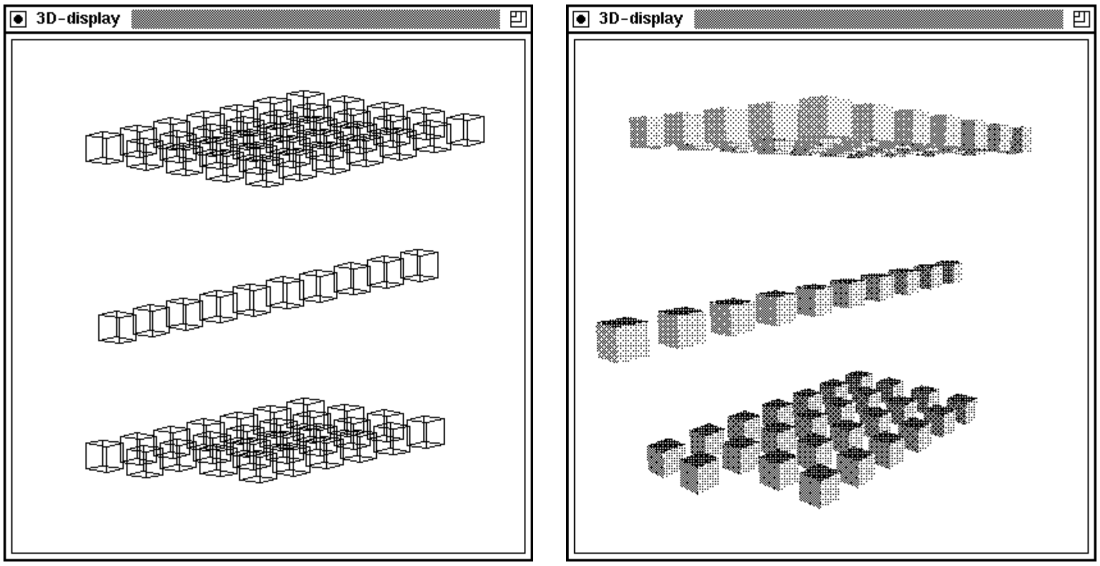

Figure 11.6: Wire-frame model in parallel projection (left) and solid model in central projection (right)

The output layer, which is assigned the z-value 6, is treated accordingly. Now the network may be rotated to any position (figure 11.6, left).

Finally the central projection and the illumination may be turned on (figure 11.6, right).

These are the links in the wire-frame model (figure 11.7, left).The network with links in the solid model looks like figure 11.7 (right).

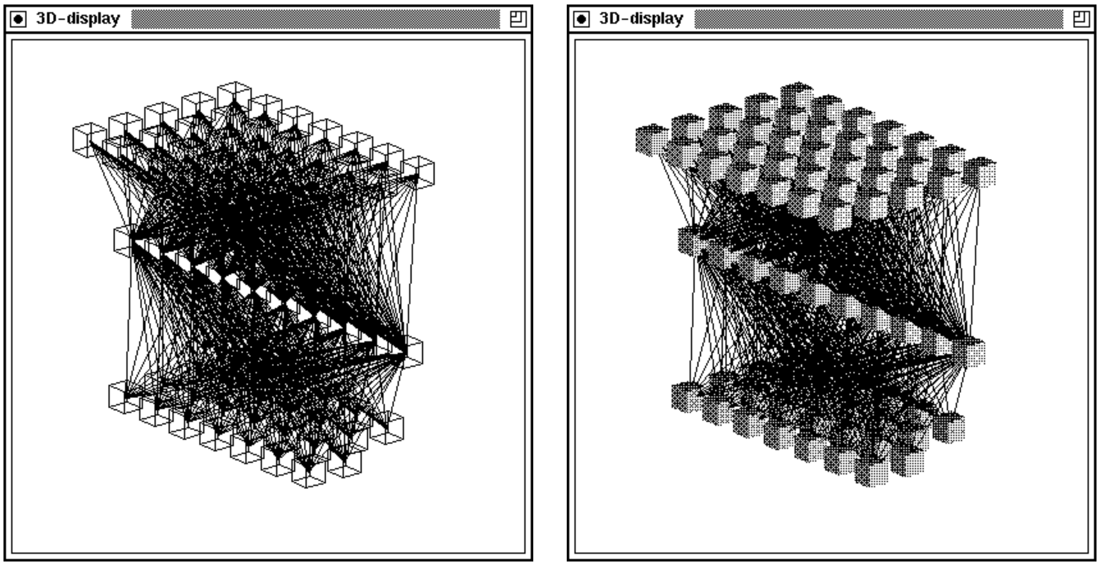

Figure 11.7: Network with links in the wireframe model (left) and in the solid model (right)

11.2.4 3D-Control Panel

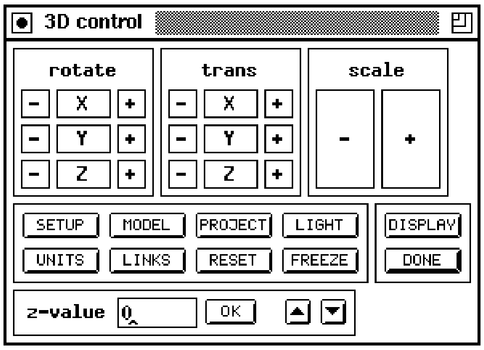

Figure 11.8: Control Panel

The 3D-control panel is used to control the display panel. It consists of four sections (panel):

1. the transformation panels

  - `rotate`: rotates the 3D-display along the x-, y- or z-axis
  - `trans`: transposes the 3D-display along the x-, y- or z-axis
  - `scale`: scales the 3D-display

2. the command panel with the buttons
   - SETUP: basic settings like rotation angles are selected
   - MODEL: switch between solid model and wireframe model
   - PROJECT: selects parallel or central projection
   - LIGHT: chooses lighting parameter
   - UNITS: selects various unit display options
   - LINKS: selects link display options
   - RESET: resets all 3D settings to their original values
   - FREEZE: freezes the 3D-display

3. the panel with the buttons
   - DISPLAY: opens the 3D-display (max. one)
   - DONE: closes the 3D-display window and the 3D-control window and exits the 3D visualization component

4. the z-value panel: used for input of z-values either directly or incrementally with the arrow buttons

11.2.4.1 Transformation Panels

With the transformation panels, the position and size of the network can be controlled.

In the `rotate` panel, the net is rotated around the x-, y-, or z-axis. The + buttons rotate clockwise, the - buttons counterclockwise. The center fields `X`, `Y` and `Z` are no buttons but framed in similar way for pleasant viewing.

In the `trans` panel, the net is moved along the x-, y-, or z-axis. The + buttons move to the right, the - buttons to the left. The center fields `X`, `Y` and `Z` are no buttons but framed in similar way for pleasant viewing.

In the `scale` panel, the net can be shrunk or enlarged.

11.2.4.2 Setup Panel

In the `base` column of the setup panel, the transformation parameters can be set explicitly to certain values. The rotation angle is given in degrees as a nine digit float number, the transposition is given in pixels, the scale factor relative to 1. Upon opening the window, the fields contain the values set by the transformation panels, or the values read from the

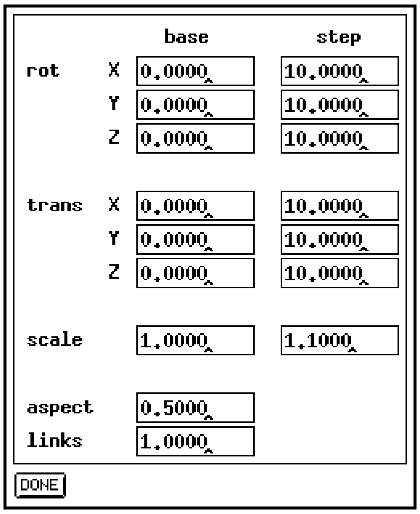

Figure 11.9: Setup Panel

configuration file. The default value for all fields is zero. The net is then displayed just as in the 2D-display.

In the `step` column the step size for the transformations can be set. The default for rotation is ten degrees, the default for moving is 10 pixel. The scaling factor is set to 1.1.

In the `aspect` field the ratio between edge length of the units and distance between units is set. Default is edge length equals distance.

With `links` the scale factor for drawing links can be set. It is set to 1.0 by default.

The `DONE` button closes the panel and redraws the net.

11.2.4.3 Model Panel

In the model panel the representation of the units is set. With the `WIRE` button a wire frame model representation is selected. The units then consist only of edges and appear transparent.

The `SOLID` button creates a solid representation of the net. Here all hidden lines are eliminated. The units' surfaces are shaded according to the illumination parameters if no other value determines the color of the units.

When the net is to be changed, the user is advised to use the wire frame model until the desired configuration is reached. This speeds up the display by an order of magnitude.

11.2.4.4 Project Panel

Here the kind of projection is selected.

`PARALLEL` selects parallel projection, i.e. parallels in the original space remain parallel.

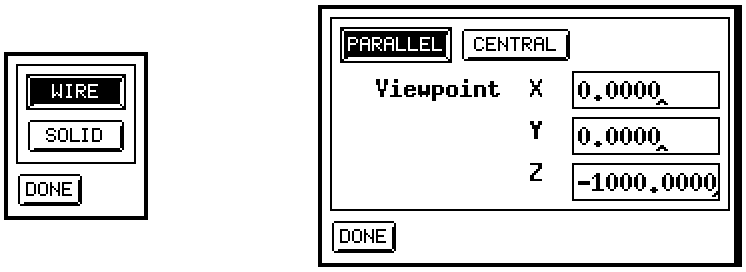

Figure 11.10: Model Panel (left) and Project Panel (right)

`CENTRAL` selects central projection, i.e. parallels in original original space intersect in the display.

With the `Viewpoint` fields, the position of the viewer can be set. Default is the point (0, 0, -1000) which is on the negative z-axis. When the viewer approaches the origin the network appears more distorted.

11.2.4.5 Light Panel

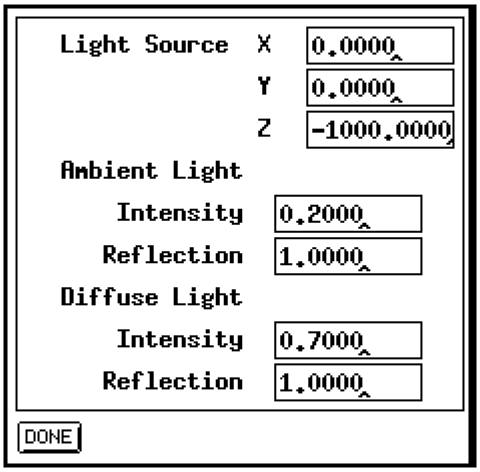

Figure 11.11: Light Panel

In the light panel, position and parameters of the light source can be set. The fields `Position` determine the location of the source. It is set to (0, 0, -1000) by default, which is the point of the viewer. This means that the net is illuminated exactly from the front. A point in positive z-range is not advisable, since all surfaces would then be shaded.

With the `Ambient Light` fields, the parameters for the background light are set.

`Intensity` sets the intensity of the background brightness.

`Reflection` is the reflection constant for the background reflection. ($0 \leq$ Ref. ≤ 1)

The fields `Diffuse Light` determine the parameters for diffuse reflection.

`Intensity` sets the intensity of the light source.

`Reflection` is the reflection constant for diffuse reflection. ($0 \leq$ Ref. ≤ 1)

11.2.4.6 Unit Panel

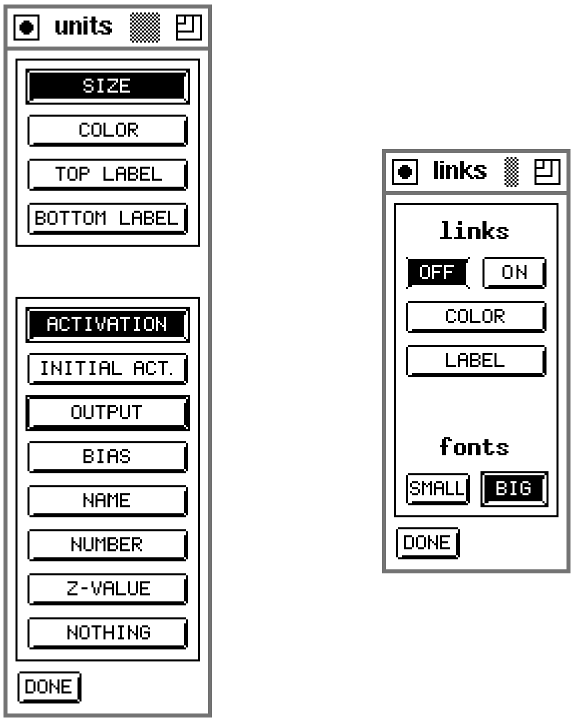

Figure 11.12: Unit Panel (left) and Link Panel (right)

With the unit panel the representation of the units can be set. The upper part shows the various properties that can be used to display the values:

- `SIZE`: a value is represented by the size of the unit. The maximum size is defined by the `Aspect` field in the setup panel. Negative and small positive values are not displayed.
- `COLOR`: a value is represented by the color of the unit. A positive value is displayed green, a negative red. This option is available only on color terminals.
- `TOP LABEL`: a value is described by a string in the upper right corner of the unit.
- `BOTTOM LABEL`: a value is described by a string in the lower right corner of the unit.

In the lower part the type of the displayed value, selected by a button in the upper part, can be set. It is displayed by

- ACTIVATION: the current activation of the unit.
- INITIAL ACT.: the initial activation of the unit.
- OUTPUT: the output value of the unit.
- BIAS: the threshold of the unit.
- NAME: the name of the unit.
- NUMBER: the number of the unit.
- Z-VALUE: the z-coordinate of the unit.
- NOTHING: no value.

The options `NAME`, `NUMBER` and `Z-value` can be used only with the top or bottom label. The other values can be combined freely, so that four values can be displayed simultaneously.

11.2.4.7 Links Panel

In the links panel the representation of the links can be switched on and off with the buttons ON and OFF. The button COLOR forces color representation of the links (only with color monitors), and the button LABEL writes the weights of the links in the middle.

In the `fonts` part of the panel, the fonts for labeling the links can be selected. The button SMALL activates the 5×8 font, the button BIG the 8×14 font

11.2.4.8 Reset Button

With the RESET button the values for moving and rotating are set to zero. The scaling factor is set to one.

11.2.4.9 Freeze Button

The FREEZE button keeps the network from being redrawn.

11.2.5 3D-Display Window

In the display window the network is shown (see figure 11.7). It has no buttons, since it is fully controlled by the control panel. It is opened by the DISPLAY button of the control panel. When the control panel is closed, the display window is closed as well.

Note: The 3D-display is only a display window, while the 2D-display windows have a graphical editor integrated. There is also no possibility to print the 3D-display via the print panel.

Chapter 12

SNNS for Workstation Clusters

Training several neural networks on a single workstation may require a large amount of training time, if performed on a single workstation. Dividing the training task among several workstations on a LAN (local area network) may significantly decrease the overall training time. This version of SNNS for workstation clusters allows to train several (possibly different) networks on different workstations and to control them with a single graphical user-interface.

The kernel interface has been extended for this purpose by an *rpc* module that is controlled by the graphical user-interface xgui. The module was named *rpc* because the communication among the workstations is performed via the rpc (remote procedure call) protocol.

This parallel SNNS version supports the following methods of parallelism:

- training different networks in parallel, each with its own pattern file,
- training several copies of the same network with different parameters or different initialization in parallel, each with its own (copied) pattern file, or
- an arbitrary combination of the above two methods.

The parallel version currently does NOT support dividing the training of a single network over several workstations. This might be added in a future version, if it can be implemented efficiently enough.

The extension consists of two programs:

- `xguirpc`: SNNS simulator program (xgui, simulator kernel, and rpc module linked into one executable program). This program is the graphical user-interface, which controls the different simulator kernels. It has the same functionality as the xgui (snns) program plus different panels to control the rpc-interface. To use multiple simulator kernels, one may start as many `kernelrpc` programs as desired.
- `kernelrpc`: simulator kernel linked with the rpc module (the server). This program is the stand alone simulator kernel with rpc module, which is controlled from the `xguirpc` program. Each kernel has is own net, pattern sets and configurations.

The files are located in the directory

<snns-root>/rpc/bin/<architecture>

where <architecture> is one of the currently supported machine architectures. We have tested it on an Intel Paragon and single CPU Sun SparcStations, and Linux systems. To use the program efficiently on multi-CPU systems like the DEC Alpha some adjustments in the libraries might be necessary.

The easiest way to start a new simulator kernel is from the **new panel** called by pressing the `NEW` button in the **SNNS rpc panel** (see 12.2.1).

If the networks are to be trained without human supervision, a similar functionality may also be obtained by starting a copy of the batch SNNS simulator `snnsbat` on each workstation.

12.1 SNNS RPC Programs

Xguirpc

`Xguirpc` has the same options as `xgui`. Additionally there is one more option:

`-force`:
The force option is instructing `xguirpc` to delete the portmap number of an old `xguirpc`. This option is only necessary if an `xguirpc` was not terminated correctly on the machine. Only the last registered `xguirpc` will receive messages from the simulator kernel. This implies that a user can run only one `xguirpc` on a computer.

Kernelrpc

`Kernelrpc` has the following options:

- `uid` <xguirpc.user-id>: the user-id of the `xguirpc` user,
- `host` <xgui_rpc_host>: the host-name of the `xguirpc`,
- `program` <xgui_program_number>: the program number under which the `xguirpc` is registered at the portmapper,
- `version` <xgui_version_number>: the version number under which the `xguirpc` is registered at the portmapper,
- `type` <type>: kernel type (normal or masters),
- `net` <net-name>: the name of a network file that is to be loaded,
- `pat` <pat-name>: the name of a pattern file that is to be loaded. This option may be given more than once.

If `kernelrpc` is started without any options the kernel will wait a short period of time. If there is no connection to an `xguirpc` the `kernelrpc` will be aborted. The default waiting time is set to 15 minutes. The simplest way to start `kernelrpc` is with the help of the **New panel** of the rpc module, because all options are set automatically. It is possible to run up to 10 `kernelrpc`s at the same time on one computer.

In some networks problems occur, caused by the domain name. If NIS (SUN Yellow pages) is activated, the function `getdomainname` does not deliver the correct domain name. However, this problem can be solved, if the environment variable DOMAINNAME is set to the correct domain name.

`Example:` On our Sun workstations `getdomainname` returns `infra` instead of `informatik.uni-stuttgart.de`. To solve this problem, we type: `setenv DOMAINNAME informatik.uni-stuttgart.de`. The best way to achieve this is to add the above line to the `.cshrc` file.

12.2 Control Panels

The distribution of SNNS onto the workstation cluster and the distributed execution of the training and recall is controlled via several windows. The main control panel, the **rpc panel**, is opened by clicking the rpc button in the manager panel.

12.2.1 SNNS RPC Panel

This panel is the control center of the communication module. In this panel the simulator kernel with which the graphical user-interface will interact is selected. Choose a row and click SWITCH TO, to specify a simulator kernel to contact. The displays and the control panel will then be updated.

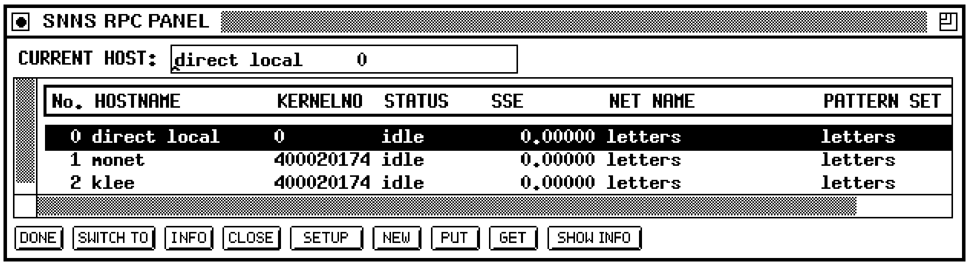

Figure 12.1: The RPC Panel

With this panel the current simulator kernel, shown in the top row, is chosen.

The buttons and fields of this panel (shown in figure 12.1) have the following functionality:

1. **CURRENT HOST** gives the name of the host computer on which the currently selected `kernelrpc` program is running.
2. The **list window** displays the list of all currently registered `kernelrpc` programs.

3. `SWITCH TO` switches to the selected simulator kernel.
4. `INFO` displays information about the selected simulator kernel. No switch of the simulator kernel takes place. The information displyed includes: Network names or comments which are saved in the network description.
5. `CLOSE` closes the selected simulator kernel. Before this process is completed, the system switches to 'direct local'. This simulator kernel is special because it is not using the standard communication protocol. It is impossible to delete this kernel because it is linked directly to the graphical user-interface. If the graphical user-interface is started, the standard simulation kernel is 'direct local'. The only difference between a 'direct local' rpc kernel and the standard SNNS kernel without rpc interface is the existence of the `RPC` button in the **Manager panel** with the added rpc functionality.
6. `SETUP` opens the **Setup panel**. In this panel the current settings of the marked kernel are displayed and can be modified. These settings are described in section 12.2.2.
7. `NEW` creates the **New panel**. With this panel it is possible to start new simulator kernels on different computers or to link already existing kernels. The **New panel** is described in section 12.2.3.
8. `PUT` and `GET` change the net and the patterns within the selected kernel.
9. `SHOW INFO` selects the information, which is displayed in the `list window`.
10. `DONE` closes the **rpc panel**.

12.2.2 RPC Setup Panel

With the help of this panel the current settings of the marked simulator kernel are displayed and modified.

- `UDP` and `TCP` switch between the connection types UDP and TCP. If the connection type is changed, the current connection is terminated and a new connection is established. `UDP` and `TCP` are the two possible protocols that are build directly upon the internet protocol (IP). `UDP` (unreliable datagram protocol) is a little bit faster than `TCP`. Since `TCP` is more reliable we recomend it for usage with `xguirpc`.
- `NORMAL` and `COOP-MASTER` set the mode of the kernel. The function of the coop-master mode has not been implemented in this release of the SNNS.
- `Standard timeout` The waiting interval of the graphical user-interface, which is used to determine when an rpc error should be displayed.
- `Long duration timeout` The long duration timeout field is responsible for the waiting time in the learning process. It specifies the time `xguirpc` will wait for a `kernlrpc` program to finish a training cycle. If there is no response within that period of time `xguirpc` will assume an errror in the program execution. This value

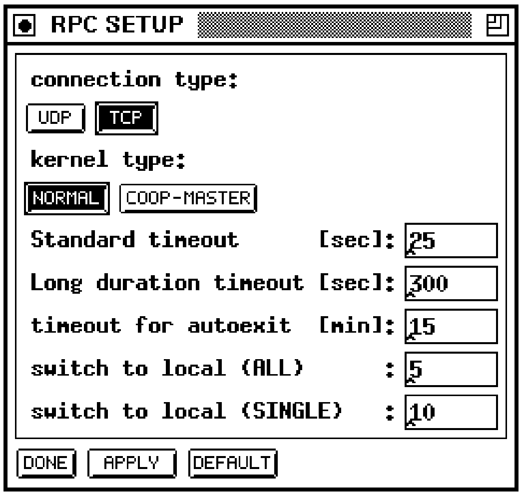

Figure 12.2: The RPC Setup Panel

should be rather large for big nets or pattern sets, otherwise an error will be displayed even if there is actually none.

- `Timeout for autoexit` is responsible to specify how long a simulator kernel will wait without receiving any messages before it automatically stops. This value is entered in minutes.
- `switch to local (ALL)` Specifies the threshold for asynchronous execution mode. If there are fewer training cycles specified in the control panel, `xguirpc` will wait for the remote host to finish learning (synchronous mode). If there are more training cycles specified, `xguirpc` will start training on the remote host and then immediately switch back to 'direct local' to allow the user further input as long as `kernlrpc` is executing training on the remote host (asynchronous mode). This value is relevant only for the training of all patterns.
- `switch to local (SINGLE)` Same as `switch to local (ALL)` but with relevance only to learning a single pattern.
- `APPLY` All entered values are set only after this button is pressed.
- `DEFAULT` will use the default values.
- `DONE` closes the window.

12.2.3 New Panel

The **New panel** is used to start new simulator kernels on different computers. A computer host name is required in the first line (it is also possible to select one from the list by clicking on it with the left mouse button). The second line specifies the path to the desired rpc kernel. Remember to seltect an rpc kernel of the correct computer architecture, if you are working with a heterogeneous workstation cluster. Different calling parameters can be added in the third line.

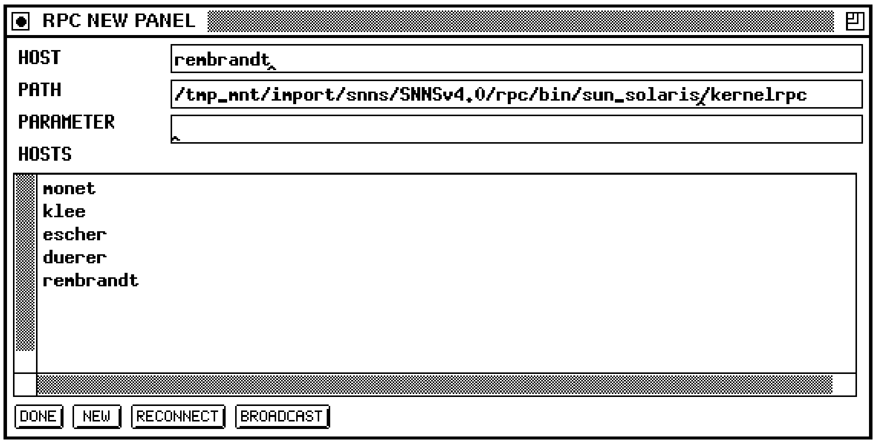

Figure 12.3: The RPC New Panel

The NEW button starts a new rpc kernel. To start the program, `xguirpc` uses the 'rsh'-instruction to execute a command on a remote computer system. The newly started simulator kernel receives the computer name, portmap number and version of the computer on which the graphical user-interface is running. After a successful start of the simulator kernel, the kernel registers with the graphical user-interface. The kernel is then included in the 'host-list' of the **rpc panel**.

If the new kernel is not listed in the 'host-list' of the **rpc panel** within a couple of seconds, it might be due to the fact that the given path is invalid on the specified machine (e.g. wrong mount prefix). In this case try to determine the correct path and try again.

With the RECONNECT and the BROADCAST buttons, different already existing simulator kernels can be connected with the graphical user-interface. The difference between those two buttons is that RECONNECT selects a specific computer and BROADCAST sends a signal to all computers in the LAN. It is not guaranteed that all computers will actually receive the signal.

12.2.4 Transfer Panels

To transfer patterns or networks to or from a local simulator kernel, the transfer panels can be used.

If the **Get** panel is chosen, it is necessary to have 'direct-local' as a current host and to have a remote simulator kernel marked with the mouse. The simulator kernel, which the nets and/or the patterns will be transfered to, can be chosen with the two arrows. Note that help files will be created in the directory /tmp as the transfer takes place. Those files will be deleted after the transfer. It is necessary to make sure that enough disk space in /tmp is available.

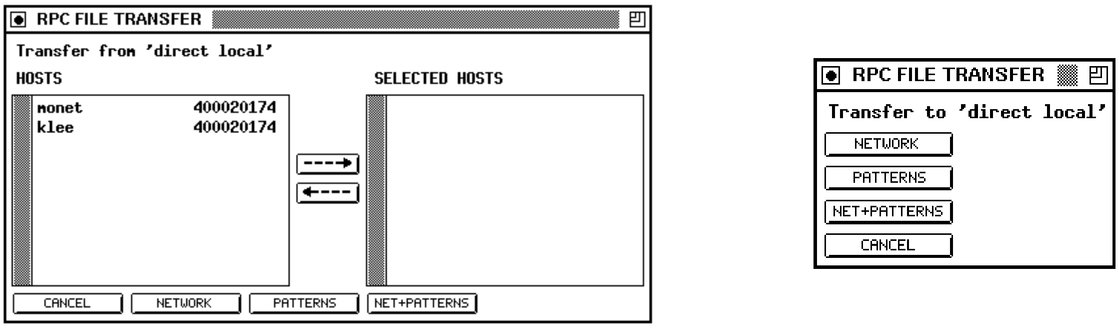

Figure 12.4: The RPC Transfer Panels. Put on the left, Get on the right.

12.2.5 Show Info Panel

With this panel it is possible to select the information, displayed in the **SNNS RPC** panel. Items can be added or deleted with the two arrows. The order of the selected items can be chosen with `UP IN LIST` or `DOWN IN LIST`.

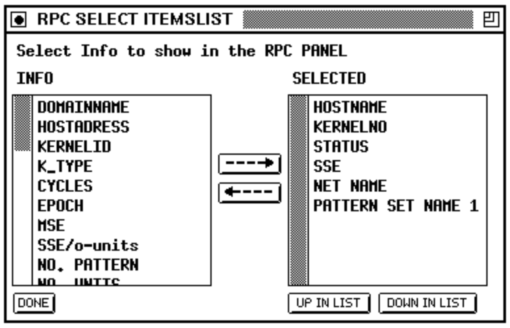

Figure 12.5: The RPC Info Panel

The possible fields are:

- `HOST-NAME`: the computer name on which the simulator kernel is running,
- `KERNEL-NAME`: the version number under which the simulator kernel is registered at the portmapper (this information can also be displayed with 'rpcinfo -p'),
- `STATUS`: the last known state of the simulator kernel,
- `DOMAINNAME`: the domain name of the network,
- `HOSTADDRESS`:the host address of the simulator kernel,
- `K-TYPE`: the type of the simulator kernel ("coop-master" and "normal" are possible, but coop-master is not implemented in the current version),
- `KERNEL_ID`: id of the simulator kernel in xgui,
- `CYCLES`: number of cycles the kernel used to train,

- `EPOCH`: number of epochs the kernel used to train,
- `SSE`: the sse error value,
- `MSE`: the mse error value,
- `SSE/o-units`: the sse error value divided bv the number of output units,
- `NO. PATTERN`: the number of patterns,
- `NO. UNITS`: the number of units,
- `NO. INPUT`: the number of input units,
- `NO. HIDDEN`: the number of hidden units,
- `NO. OUTPUT`: the number of output units,
- `NET NAME`: the network name,
- `PATTERN SET NAME`: the name of the pattern sets,
- `LEARN`: the selected learning parameters,
- `UPDATE`: the selected update parameters,
- `INIT`: the selected initialization parameters.

Note that all information is usually not displayed immediately, but with a small delay.

12.3 RPC Errors

If an rpc function is called, there may be additional errors besides regular SNNS errors, such as the call of the `DeletePattern` function without a defined pattern in the network. Because `rpcgen` is not generating error code within the rpc routines the result of these functions has to be checked for errors. Since these routines return a NULL value in case of an error, the return value can be checked for NULL and the system can then call an error routine, which determines the exact cause of the error. The rpc library provides such an error routine. Because the rpc mechanism can not determine if there is a net-adapter malfunction or if one of the cables is defect, the rpc mechanism only determines that there is no response from anyone within a given time interval. The error cause, delivered by the error function is not that an adapter card is defect, but that the time interval was exceeded (time-out error). This is the most common type of error and the communication module tries to detect the error with the help of the graphical user-interface.

To accomplish such a task, the communication module remembers if the last inquiry was performed synchronously or asynchronously. If the inquiry was asynchronous the probable cause is that the server is not ready for a new task, because it has to finish the current one. Therefore the simulator kernel reports to the graphical user-interface as soon as it has finished its task. This enables the communication module to update all status informations about the simulator kernel. The user will be informed about an error, and if available what kind of error, over a ‘pop up’ window.

Due to the separation of the graphical user-interface and the simulator kernel, the error tolerance has increased. If there is for example an error in the graphical user-interface, only the interface will be terminated and after a new start of the graphical user-interface and its linking to the simulator kernel, the user is able to proceed exactly where the error occured.

If an error occurs in the kernel the situation is less complicated. In that case simply a new net has to be started, since the communication module intercepts all maskable signals and saves the current net before it ends its operation correctly.

This kind of independence can be used to train multiple nets in parallel. All that has to be done is to load multiple nets with multiple simulator kernels and to start the training process asynchronously with the desired parameters. Now the graphical user-interface can be terminated, while the simulator kernel trains the nets. It is possible to interrupt the training and to evaluate the current state of the training process with the graphical user-interface. After that, the training can be resumed. In case the simulator kernel has already finished training the net it will wait a given period of time before it saves the net and ends its operations. In contrast to the batch version of SNNS, `snnsbat`, it is possible to select all actions of the kernel from a menu and to evaluate the current training, e.g. the current epochs and error rate.

Chapter 13

Running SNNS as Batch Job

Since training a neural network may require several hours of CPU time, it is advisable to perform this task as a batch job during low usage times. SNNS offers the program `snnsbat` for this purpose. It is basically an additional interface to the kernel that allows easy background execution. Its flexible setup allows for a variety of possible execution modes which can be easily defined in a configuration file. All actions and messages are recorded in a log file for later verification of results.

13.1 The Snnsbat Environment

`snnsbat` runs very dependably even on unstable system configurations and is secured against data loss due to system crashes, network failures etc.. On UNIX based systems the program may be terminated with the command 'kill -15' without loosing the currently computed network.

The calling syntax of snnsbat is:

snnsbat [< configuration_file > [< log_file >]]

This call starts `snnsbat` in the foreground. On UNIX systems the command for background execution is 'at', so that the command line

echo 'snnsbat default.cfg log.file' | at 22:00

would start the program tonight at 10pm[1].

If the optional file names are omitted, `snnsbat` tries to open the configuration file './snnsbat.cfg' and the protocol file './snnsbat.log'.

13.2 Using Snnsbat

The batch mode execution of SNNS is controlled by the configuration file. It contains entries that define the network and parameters required for program execution. These

[1] This construction is necessary since 'at' can read only from stdin.

entries are tuples (mostly pairs) of a keyword followed by one or more values. There is only one tuple allowed per line, but lines may be separated by an arbitrary number of comment lines. Comments start with the number sign '#'. The set of given tuples specify the actions performed by SNNS in one execution run. An arbitrary number of execution runs can be defined in one configuration file, by separating the tuple sets with the keyword 'PerformActions:'. Within a tuple set, the tuples may be listed in any order. If a tuple is listed several times, values that are already read are overwritten. The only exception is the key 'Type:', which has to be listed only once and as the first key. If a key is omitted, the corresponding value(s) are assigned a default.

Here is a listing of the tuples and their meaning:

Key	Value	Meaning
InitFunction:	<string>	Name of the initialization function.
InitParam:	<float> ···	'NoOfInitParam' parameters for initialization function, separated by blanks.
LearnParam:	<float> ···	'NoOfLearnParam' parameters for learning function, separated by blanks.
UpdateParam:	<float> ···	'NoOfUpdateParam' parameters for the update function, separated by blanks.
LearnPatternFile:	<string>	Filename of the learning patterns.
MaxErrorToStop:	<float>	Network error when learning is to be halted.
MaxLearnCycles:	<int>	Maximum number of learning cycles to be executed.
NetworkFile:	<string>	Filename of the net to be trained.
NoOfInitParam:	<int>	Number of parameters for the initialization function.
NoOfLearnParam:	<int>	No of parameters for learning function.
NoOfUpdateParam:	<int>	No of parameters for update function.
NoOfVarDim:	<int> <int>	Number of variable dimensions of the input and output patterns.
PerformActions:	none	Execution run separator.
PruningMaxRetrainCycles:	<int>	maximum no. of cycles per retraining
PruningMaxErrorIncrease:	<float>	Percentage to be added to the first net error. The resulting value cannot be exceeded by the net error, unless it is lower than the accepted error
PruningAcceptedError:	<float>	Maximum accepted error.
PruningRecreate:	[YES | NO]	Flag for reestablishing the last state of the net at the end of pruning
PruningOBSInitParam:	<float>	initial value for OBS
PruningInputPruning:	[YES | NO]	Flag for input unit pruning
PruningHiddenPruning:	[YES | NO]	Flag for hidden unit pruning
ResultFile:	<string>	Filename of the result file.

Key	Value	Meaning
ResultIncludeInput:	[YES | NO]	Flag for inclusion of input patterns in the result file.
ResultIncludeOutput:	[YES | NO]	Flag for inclusion of output learning patterns in the result file.
SubPatternOSize:	<int> ···	NoOfVarDim[2] int values that specify the shape of the sub patterns of each output pattern.
SubPatternOStep:	<int> ···	NoOfVarDim[2] int values that specify the shifting steps for the sub patterns of each output pattern.
TestPatternFile:	<string>	Filename of the test patterns.
TrainedNetworkFile:	<string>	Filename where the net should be stored after training / initialization.
Type:	<string>	The type of grammar that corresponds to this file. Valid types are: 'SNNSBATCH_1': performs only one execution run. 'SNNSBATCH_2': performs multiple execution runs.
ResultMinMaxPattern:	<int> <int>	Number of the first and last pattern to be used for result file generation.
Shuffle:	[YES | NO]	Flag for pattern shuffling.
ShuffleSubPat:	[YES | NO]	Flag for subpattern shuffling.
SubPatternISize:	<int> ···	NoOfVarDim[1] int values that specify the shape of the sub patterns of each input pattern.
SubPatternIStep:	<int> ···	NoOfVarDim[1] int values that specify the shifting steps for the sub patterns of each input pattern.

Please note the mandatory colon after each key and the upper case of several letters.

`snnsbat` may also be used to perform only parts of a regular network training run. If the network is not to be initialized, training is not to be performed, or no result file is to be computed, the corresponding entries in the configuration file can be omitted.

For all keywords the string '<OLD>' is also a valid value. If <OLD> is specified, the value of the previous execution run is kept. For the keys 'NetworkFile:' and 'LearnPatternFile:' this means, that the corresponding files are not read in again. The network (patterns) already in memory are used instead, thereby saving considerable execution time. This allows for a continuous logging of network performance. The user may, for example, load a network and pattern file, train the network for 100 cycles, create a result file, train another 100 cycles, create a second result file, and so forth. Since the error made by the current network in classifying the patterns is reported in the result file, the series of result files document the improvement of the network performance.

The following table shows the behavior of the program caused by omitted entries:

missing key	resulting behavior
InitFunction:	The net is not initialized.
InitParam:	Init function gets only zero values as parameters.
LearnParam:	Learning function gets only zero values as parameters.
UpdateParam:	Update function gets only zero values as parameters.
LearnPatternFile:	Abort with error message if more than 0 learning cycles are specified. Initialization can be performed if init function does not require patterns.
MaxErrorToStop:	Training runs for 'MaxLearnCycles:' cycles.
MaxLearnCycles:	No training takes place. If training is supposed to run until *MaxErrorToStop*, a rather huge number should be supplied here. (skipping this entry would inhibit training completely).
NetworkFile:	Abort with error message.
NoOfInitParam:	No parameters are assigned to the initialization function. Error message from the SNNS kernel possible.
NoOfLearnParam:	No parameters are assigned to the learning function. Error message from the SNNS kernel possible.
NoOfUpdateParam:	No parameters are assigned to the update function.
NoOfVarDim:	Network can not handle variable pattern sizes.
PerformActions:	Only one execution run is performed. Repeated keywords lead to deletion of older values.
ResultFile:	No result file is generated.
ResultIncludeInput:	The result file does NOT contain input Patterns.
ResultIncludeOutput:	The result file DOES contain learn output Patterns.
ResultMinMaxPattern:	All patterns are propagated.
Shuffle:	Patterns are not shuffled.
ShuffleSubPat:	Subpatterns are not shuffled.
SubPatternISize:	
SubPatternIStep:	
SubPatternOSize:	
SubPatternOStep:	Abort with error message if 'NoOfVarDim:' was specified.
TestPatternFile:	Result file generation uses the learning patterns. If they are not specified either, the program is aborted with an error message when trying to generate a result file.
TrainedNetworkFile:	Network is not saved after training / initialization. It is used for result file generation.
Type:	Abort with error message.

Here is a typical example of a configuration file:

```
#
Type: SNNSBATCH_2
#
# If a key is given twice, the second appearance is taken.
# Keys that are not required for a special run may be omitted.
# If a key is omitted but required, a default value is assumed.
# The lines may be separated with comments.
#
# Please note the mandatory file type specification at the beginning and
# the colon following the key.
#
#######################################################################
NetworkFile: /home/SNNSv4.0/examples/letters.net
#
InitFunction: Randomize_Weights
NoOfInitParam: 2
InitParam: -1.0 1.0
#
LearnPatternFile: /home/SNNSv4.0/examples/letters.pat
NoOfVarDim: 2 1
SubPatternISize: 5 5
SubPatternOSize: 26
SubPatternIStep: 5 1
SubPatternOStep: 1
NoOfLearnParam: 2
LearnParam: 0.8 0.3
MaxLearnCycles: 100
MaxErrorToStop: 1
Shuffle: YES
#
TrainedNetworkFile: trained_letters.net
ResultFile: letters1.res
ResultMinMaxPattern: 1 26
ResultIncludeInput: NO
ResultIncludeOutput: YES
#
#This execution run loads a network and pattern file with variable
#pattern format, initializes the network, trains it for 100 cycles
#(or stops, if then error is less than 0.01), and finally computes
#the result file letters1.
PerformActions:
#
NetworkFile: <OLD>
#
LearnPatternFile: <OLD>
NoOfLearnParam: <OLD>
LearnParam: <OLD>
MaxLearnCycles: 100
MaxErrorToStop: 1
Shuffle: YES
#
ResultFile: letters2.res
ResultMinMaxPattern: <OLD>
ResultIncludeInput:  <OLD>
ResultIncludeOutput: <OLD>
```

```
#
#This execution run continues the training of the already loaded file
#for another 100 cycles before creating a second result file.
#
PerformActions:
#
NetworkFile: <OLD>
#
LearnPatternFile: <OLD>
NoOfLearnParam: <OLD>
LearnParam: 0.2 0.3
MaxLearnCycles: 100
MaxErrorToStop: 0.01
Shuffle: YES
#
ResultFile: letters3.res
ResultMinMaxPattern: <OLD>
ResultIncludeInput:  <OLD>
ResultIncludeOutput: <OLD>
TrainedNetworkFile: trained_letters.net
#
#This execution run concludes the training of the already loaded file.
#After another 100 cycles of training with changed learning
#parameters the final network is saved to a file and a third result
#file is created.
#
```

The file <log_file> collects the SNNS kernel messages and contains statistics about running time and speed of the program. An example protocol file is listed in appendix C.

If the <log_file> command line parameter is omitted, `snnsbat` opens the file 'snnsbat.log' in the current directory. To limit the size of this file, a maximum of 100 learning cycles are logged. This means, that for 1000 learning cycles a message will be written to the file every 10 cycles.

If the time required for network training exceeds 30 minutes of CPU time, the network is saved. The log file then shows the message:

```
#####  Temporary network file 'SNNS_Aaaa00457' created.  #####
```

Temporay networks always start with the string 'SNNS_'. After 30 more minutes of CPU time, `snnsbat` creates a second security copy. Upon normal termination of the program, these copies are deleted from the current directory. The log file then shows the message:

```
#####  Temporary network file 'SNNS_Aaaa00457' removed.  #####
```

In an emergency (powerdown, kill, alarm, etc.), the current network is saved by the program. The log file, resp. the mailbox, will later show an entry like:

```
Signal 15 caught, SNNS V4.0 Batchlearning terminated.

SNNS V4.0 Batchlearning terminated at Tue Mar 23 08:49:04 1993
System:  SunOS Node:  matisse Machine:  sun4m
Networkfile './SNNS_BAAa02686' saved.
Logfile 'snnsbat.log' written.
```

13.3 Calling Snnsbat

`snnsbat` may be called interactively or in batch mode. It was designed, however, to be called in batch mode. On Unix machines, the command 'at' should be used, to allow logging the program with the mailbox. However, 'at' can only read from standard input, so a combination of 'echo' and 'pipe' has to be used.

Three short examples for Unix are given here, to clarify the calls:

unix>echo 'snnsbat mybatch.cfg mybatch.log' | at 21:00 Friday

starts `snnsbat` next Friday at 9pm with the parameters given in mybatch.cfg and writes the output to the file mybatch.log in the current directory.

unix>echo 'snnsbat SNNSconfig1.cfg SNNSlog1.log' | at 22

starts `snnsbat` today at 10pm

unix>echo 'snnsbat' | at now + 2 hours

starts `snnsbat` in 2 hours and uses the default files snnsbat.cfg and snnsbat.log

The executable is located in the directory '.../SNNSv4.0/kernel/bin/<machine_type>/'.
The sources of snnsbat can be found in the directory '.../SNNSv4.0/kernel/sources/'.
An example configuration file was placed in '.../SNNSv4.0/examples'.

Chapter 14

Tools for SNNS

14.1 Overview

There are the following tools available to ease the use of SNNS:

`analyze:`	analyzes result files generated by SNNS to test the classification capabilities of the corresponding net
`bignet:`	feedforward network generator
`Convert2snns:`	pattern conversion tool for Kohonen Networks
`feedback-gennet:`	generator for network definition files
`mkhead:`	writes SNNS pattern file header to stdout
`mkout:`	writes SNNS output pattern to stdout
`mkpat:`	reads 8 bit rawfile and writes SNNS pattern file to stdout
`netlearn:`	backpropagation test program
`netperf:`	benchmark program
`pat_sel:`	produces pattern file with selected patterns
`snns2c:`	compiles an SNNS network file into an executable C source.

14.2 Analyze

The purpose of this tool is to analyze the result files that have been created by SNNS. The result file which you want to analyze has to contain the `teaching output` and the `output of the network`.

Synopsis: `analyze` [-options]

It is possible to choose between the following options in any order:

`-w`	numbers of patterns which were classified wrong are printed
`-r`	numbers of patterns which were classified right are printed
`-u`	numbers of patterns which were not classified are printed
`-a`	same as -w -r -u

`-v`	verbose output. Each printed number is preceded by one of the words 'wrong', 'right' or 'unknown' depending on the result of the classification.
`-s`	statistic information containing wrong, right and not classified patterns. The network error is printed also.
`-c`	same as -s, but statistics for each output unit (class) is displayed.
`-i <file name>`	name of the 'result file' which is going to be analyzed.
`-o <file name>`	name of the file which is going to be produced by `analyze`.
`-e <function>`	defines the name of the 'analyzing function'. Possible names are: 402040, WTA, band (description see below)
`-l <real value>`	first parameter of the analyzing function.
`-h <real value>`	second parameter of the analyzing function.

Starting `analyze` without any options is equivalent to:

```
analyze -w -e 402040 -l 0.4 -h 0.6
```

14.2.1 Analyzing Functions

The classification of the patterns depends on the analyzing function. `402040` stands for the '402040' rule. That means on a range from 0 to 1 h will be 0.6 (upper 40%) and l will be 0.4 (lower 40%). The middle 20% is represented by h-l. The classification of the patterns will depend on h, l and other constrains (see `402040` below).
`WTA` stands for winner takes all. That means the classification depends on the unit with the highest output and other constrains (see `WTA` below). `Band` is an analyzing function that checks a band of values around the teaching output. This is the only analyzing function that treats teaching outputs of value zero correctly.

402040:
A pattern is classified correctly if:

- the output of exactly one output unit is $\geq$ h.
- the 'teaching output' of this unit is the maximum teaching output of the pattern.
- the output of all other output units is $\leq$ l.

A pattern is classified incorrectly if:

- the output of exactly one output unit is $\geq$ h.
- the 'teaching output' of this unit is `NOT` the maximum 'teaching output' of the pattern.
- the output of all other units is $\leq$ l.

A pattern is unclassified in all other cases.

WTA:
A pattern is classified correctly if:

- there is an output unit with the value greater than the output value of all other output units (this output value is supposed to be 0).
- 0 > h.
- the 'teaching output of this unit is the maximum 'teaching output' of the pattern.
- the output of all other units is < a-1.

A pattern is classified incorrectly if:

- there is an output unit with the value greater than the output value of all other output units (this output value is supposed to be 0).
- 0 > h.
- the 'teaching output ' of this unit is `NOT` the maximum 'teaching output' of the pattern.
- the output of all other output units is < a-1.

A pattern is unclassified in all other cases.

Band:
A pattern is classified correctly if for all output units:

- the output is >= the teaching output - low.
- the output is <= the teaching output + high.

A pattern is classified incorrectly if for all output units:

- the output is <= the teaching output - low.
  or
- the output is >= the teaching output + high.

Default values used for l and h are: l=0.0 h=0.0

14.3 Bignet

This program is a generator for special 3-layered feedforward networks. It is a demo for using the SNNS kernel interface to create networks. It has nothing to do with the XGUI bignet tool.

Synopsis: `bignet`

example:

`unix>` bignet

produces

SNNS 3D-Kernel 4.0	— Network Generator for 3 Layer Feedforward Networks —
No. of input units:	35
No. of output units:	26
No. of hidden units:	10

Create Units now...
make connections

Name of the network :	test
Filename of the network to save:	test.net
Saving the network...	

14.4 Convert2snns

In order to work with the KOHONEN tools in SNNS, a pattern file and a network file with a special format are necessary.

Convert2snns will accomplish three important things:

- Creation of a 2-dimensional Kohonen Feature Map with n components
- Weight files are converted in a SNNS compatible .net file
- A file with raw patterns is converted in a .pat file

When working with convert2snns, 3 files are necessary:

1. A control file, containing the configuration of the network
2. A file with weight vectors
3. A file with raw patterns

14.4.1 Setup and Structure of a Control, Weight, Pattern File

Each line of the control file begins with a `KEYWORD` followed by the respective declaration. The order of the keywords is arbitrary.

Example of a control file:

`PATTERNFILE` eddy.in	**
`WEIGHTFILE` eddy.dat	
`XSIZE` 18	←
`YSIZE` 18	←
`COMPONENTS` 8	←
`PATTERNS` 47	**

For creation of a network file you need at least the statements marked ← and for the .pat file additionally the statements marked **.
Omitting the `WEIGHTFILE` will initialize the weights of the network with 0.

The `WEIGHTFILE` is a simple ASCII file, containing the weight vectors row by row.

The `PATTERNFILE` contains in each line the components of a pattern.

If convert2snns has finished the conversion it will ask for the name of the network and pattern files to be saved.

14.5 Feedback-gennet

The program `feedback-gennet` generates network definition files for fully recurrent networks of any size. This is not possible by using bignet.

The networks have the following structure:

- input layer with no intra layer connections
- fully recurrent hidden layer
- output layer: connections from each hidden unit to each output unit

`AND`

optionally fully recurrent intra layer connections in the output layer

`AND`

optionally feedback connections from each output unit to each hidden unit.

The activation function of the output units can be set to sigmoidal or linear. All weights are initialized with 0.0. Other initializations should be performed by the init functions in SNNS.

Synopsis: `feedback-gennet`

example:

```
unix> feedback-gennet
```

produces

```
Enter # input units:   3
Enter # hidden units:  3
Enter # output units:  1
INTRA layer connections in the output layer        (y/n) :n
feedback connections from output to hidden units   (y/n) :n
Linear output activation function                  (y/n) :n
Enter name of the network file: xor-rec.net
working...
generated xor-rec.net
```

14.6 Mkhead

This program writes a SNNS pattern file header to stdout. This program can be used together with mkpat and mkout to produce pattern files from raw files in a shell script.

Synopsis: `mkhead` <pats> <in_units> <out_units>

where:

`pats`	are the number of patterns in the file
`in_units`	are the number of input units in the file
`out_units`	are the number of output units in the file

14.7 Mkout

This program writes a SNNS output pattern to stdout. This program can be used together with mkpat and mkhead to produce pattern files from raw files in a shell script.

Synopsis: `mkout` <units> <active_unit>

where:

`units`	is the number of output units
`active_unit`	is the unit which has to be activated

14.8 Mkpat

The purpose of this program is to read a binary 8-Bit file from the stdin and writes a SNNS pattern file entry to stdout. This program can be used together with mkpat and mkout to produce pattern files from raw files in a shell script.

Synopsis: `mkpat` <xsize> <ysize>

where:

`xsize`	is the xsize of the raw file
`ysize`	is the ysize of the raw file

14.9 Netlearn

This is a SNNS kernel backpropagation test program. It is a demo for using the SNNS kernel interface to train networks.

Synopsis: `netlearn`

example:

unix> netlearn

produces

```
SNNS 3D-Kernel V 4.0
—Network learning—

Filename of the network file: letters_untrained.net
Loading the network...

Network name: letters
No. of units:           71
No. of input units:     35
No. of output units:    26
No. of sites:           0
No. of links:           610

Learning function:      Std_Backpropagation
Update function:        Topological_Order

Filename of the pattern file: letters.pat
loading the patterns...

Number of pattern:      26

The learning function Std_Backpropagation needs 2 input parameters:
Parameter [1]:          0.6
Parameter [2]:          0.6
Choose number of cycles: 250

Shuffle patterns (y/n)  n
Shuffling of patterns disabled

learning...
```

14.10 Netperf

This is a benchmark program for SNNS. Propagation and backpropagation tests are performed.

Synopsis: `netperf`

example:

unix> netperf

produces

SNNS 3D-Kernel V4.0	— Benchmark Test —
Filename of the network file: nettalk.net	
loading the network...	Network name: nettalk1
No. of units:	349
No. of input units:	203
No. of ouput units:	26
No. of sites:	0
No. of links:	27480
Learningfunction:	Std_Backpropagation
Updatefunction:	Topological_Order
Do you want to benchmark	
Propagation	[1] or
Backpropagation	[2]
Input:	1
Choose no. of cycles	
Begin propagation...	
No. of units updated:	34900
No. of sites updated:	0
No. of links updated:	2748000
CPU Time used:	3.05 seconds
No. of connections per second (CPS) : 9.0099e+05	

14.11 Pat_sel

Given a pattern file and a file which contains numbers, `pat_sel` produces a new pattern file which contains the subset of the first one. This pattern file consists of the patterns whose numbers are given in the number file.

Synopsis: `pat_sel` <number file> <input pattern file> <output pattern file>

Parameters:

`<number file>`	ASCII file which contains positive integer numbers (one per line) in ascending order.
`<input pattern file>`	SNNS pattern file.
`<output pattern file>`	SNNS pattern file which contains the selected subset (created by `pat_sel`)

`Pat_sel` can be used to create a pattern file which contains only the patterns that were classified 'wrong' by the neural network. That is why a 'result file' has to be created using

SNNS. The result file can be analyzed with the tool analyze. This 'number file' and the corresponding 'pattern file' are used by pat_sel. The new 'pattern file' will be created.

14.12 Snns2c

Synopsis: `snns2c` <network> [<C-filename> [<function-name>]]

where:

`<network>`	is the name of the SNNS network file,
`<C-filename>`	is the name of the output file
`<function-name>`	is the name of the procedure in the application.

This tool compiles an SNNS network file into an executable C source. It reads a network file `<network.net>` and generates a C source named `<C-filename>`. The network can be called now as a function named `<function-name>`. If the parameter `<function-name>` is missing, the name of `<C-filename>` is taken without the ending "*.c". If this parameter is also missing, the name of the network file is chosen and fitted with a new ending for the output file. This name without ending is also used for the function name.

It is not possible to train the generated net, SNNS has to be used for this purpose. After completion of network training with SNNS, the tool SNNS2C is used to integrate the trained network as a C function into a separate application.

This program is also an example how to use the SNNS kernel interface for loading a net and changing the loaded net into another format. All data and all SNNS functions – except the activation functions – are placed in a single C function.

Note:
SNNS2C does not support sites. Any networks created with SNNS that make use of the site feature can not be converted to C source by this tool. Output functions are not supported, either.

The program can translate the following network-types:

- Feedforward networks trained with Backpropagation and all variants of it like Quickprop, RPROP etc.
- Radial Basis Functions
- Partially-recurrent Elman and Jordan networks
- Time Delay Neural Networks (TDNN)
- Dynamic Learning Vector Quantisation (DLVQ)
- Backpropagation Through Time (BPTT, QPTT, BBPTT)
- Counterpropagation Networks

While the use of SNNS or any parts of it in commercial applications requires a special agreement/licensing from the developers, the use of trained networks generated with

SNNS2C is hereby granted without any fees for any purpose, provided proper academic credit to the SNNS team is given in the documentation of the application.

14.12.1 Program Flow

Because the compilation of very large nets may require some time, the program outputs messages, indicating which state of compilation is passed at the moment.

loading net... the network file is loaded with the function offered by the kernel user interface.

dividing net into layers ... all units are grouped into layers where all units have the same type and the same activation function. There must not exist any dependencies between the units of the layers except the connections of SPECIAL HIDDEN units to themselves in Elman and Jordan networks or the links of the BPTT-networks.

sorting layers... these layers are sorted in topological order, e.g. first the input layer, then the hidden layers followed by the output layers and at last the special hidden layers. A layer which has sources in another layer of the same type is updated later as the source layer.

writing net... selects the needed activation functions and writes them to the C-source file. After that, the procedure for pattern propagation is written.

14.12.2 Including the Compiled Network in the Own Application

Interfaces: All generated networks may be called as C functions. This functions have the form:

`int`*`function-name`*`(float *in, float *out, int init)`

where `in` and `out` are pointers to the input and output arrays of the network. The init flag is needed by some network types and it's special meaning is explained in 14.12.3. The function normally returns the value 0 (OK). Other return values are explained in section 14.12.3.

The generated C-source can be compiled separately. To use the network it's necessary to include the generated header file (*.h) which is also written by SNNS2C. This header file contains a prototype of the generated function and a record which contains the number of input and output units also.

Example: If a trained network, was saved as “myNetwork.net” and compiled with

```
snns2c myNetwork.net
```

then the generated Network can be compiled with

```
gcc -c myNetwork.c
```

To include the network in your own application the header file must be included. There should also two arrays being provided, one for the input and one for the output of the network. The number of inputs and outputs can be derived from a record in the header file. This struct is named like the function which contains the compiled network and has the suffix REC to mark the record. So the number of input units is determined with `myNetworkREC.NoOfInput` and the number of outputs with `myNetworkREC.NoOfOutput` in this example. Hence, your own application should contain:

```
⋮

#include "myNetwork.h"

⋮

float *netInput, *netOutput; /* Input and Output arrays of the Network */
netInput  = malloc(myNetworkREC.NoOfInput  * sizeof(float));
netOutput = malloc(myNetworkREC.NoOfOutput * sizeof(float));

⋮

myNetwork(netInput, netOutput, 0)

⋮
```

Don't forget to link the object code of the network to your application

14.12.3 Special Network Architectures

Normally, the architecture of the network and the numbers of the units are kept. Therefore a dummy unit with the number 0 is inserted in the array which contains the units. Some architectures are translated with other special features.

TDNN: Generally, a layer in a time delay neural network consists of feature units and their delay units. SNNS2C generates code only containing the feature units. The delay units are only additional activations in the feature unit. This is possible because every delay unit has the same link weights to it's corresponding source units as its feature unit.

So the input layer consists only of its prototype units, too. Therefore it's not possible to present the whole input pattern to the network. This is not necessary because it can be presented step by step to the inputs. This is useful for a real-time application, with the newest feature units as inputs. To mark a new sequence the *init*-flag (parameter of the function) can be set to 1. After this, the delays are filled when the init flag is set to 0 again. To avoid meaningless outputs the function returns NOT_VALID until the delays are filled again.

There is a new variable in the record of the header file for TDNNs. It is called "MinDelay" and is the minimum number of time steps which are needed to get a valid output after the *init*-flag was set.

CPN: Counterpropagation doesn't need the output layer. The output is calculated as a weighted sum of the activations of the hidden units. Because only one hidden unit has the activation 1 and all others the activation 0, the output can be calculated with the winner unit, using the weights from this unit to the output.

DLVQ: Here no output units are needed either. The output is calculated as the bias of the winner unit.

BPTT: If all inputs are set to zero, the net is not initialized. This feature can be chosen by setting the *init*-flag.

14.12.4 Activation Functions

Supported Activation Functions: Following activation functions are implemented in SNNS2C

Supported Activation Functions	
Act_Logistic	Act_StepFunc
Act_Elliott	Act_Identity
Act_BSB	Act_IdentityPlusBias
Act_TanH	Act_RBF_Gaussian
Act_TanHPlusBias	Act_RBF_MultiQuadratic
Act_TanH_Xdiv2	Act_RBF_ThinPlateSpline
Act_Perceptron	Act_TD_Logistic
Act_Signum	Act_TD_Elliott
Act_Signum0	

Including Own Activation Functions: The file "tools/sources/functions.h" contains two arrays: One array with the function names (ACT_FUNC_NAMES) and one for the macros which represent the function (ACT_FUNCTIONS). These macros are realized as character strings so they can be written to the generated C-source.

The easiest way to include an own activation function is to write the two necessary entries in the first position of the arrays. After that the constant "ActRbfNumber" should be increased. If a new Radial Basis function should be included, the entries should be appended at the end without increasing ActRbfNumber. An empty string ("") should still be the last entry of the array Act_FUNC_NAMES because this is the flag for the end of the array.

14.12.5 Error Messages

Here is the list of possible error messages and their brief description.

not enough memory: a dynamic memory allocation failed.

Note:
The C-code generated by snns2c that describes the network (units and links) defines

one unit type which is used for all units of the network. Therefore this unit type allocates as many link weights as necessary for the network unit with the most input connections. Since this type is used for every unit, the necessary memory space depends on the number of units times size of the biggest unit.

In most cases this is no problem when you use networks with moderate sized layers. But if you use a network with a very large input layer, your computer memory may be to small. For example, an $n - m - k$ network with $n > m > k$ needs about $n * (n + m + k) * (sizeof(float) + sizeof(void*))$ bytes of memory, so the necessary space is of $O(n^2)$ where n is the number of units of the biggest layer.

We are aware of this problem and will post an improved version of snns2c as soon as possible.

can't load file: SNNS network-file wasn't found.

can't open file: same as can't load or disk full.

wrong parameters: wrong kind or number of parameters.

net contains illegal cycles: there are several possibilities:

- a connection from a unit which is not a SPECIAL HIDDEN unit to itself.
- two layers are connected to each other when not exactly one of them is SPECIAL HIDDEN.
- cycles over more than two layers which don't match the architecture of Jordan networks.

BPTT-networks have no restrictions concerning links.

can't find the function *actfunc*: the activation function *actfunc* is not supported.

net is not a CounterPropagation network: Counterpropagation-networks need a special architecture: one input, one output and one hidden layer which are fully connected.

net is not a Time Delay Neural Network: The SNNS TDNNs have a very special architecture. They should be generated by the BIGNET-tool. In other cases there is no guaranty for successful compilation.

not supported network type: There are several network types which can't be compiled with the SNNS2C. The SNNS2C-tool will be maintained and so new network types will be implemented.

unspecified Error: should not occur, it's only a feature for user defined updates.

unknown error code : *number*: must not occur, please report to the SNNS-Team.

Chapter 15

Kernel Function Interface

15.1 Overview

The simulator kernel offers a variety of functions for the creation and manipulation of networks. These can roughly be grouped into the following categories:

- functions to manipulate the network
- functions to determine the structure of the network
- functions to define and manipulate cell prototypes
- functions to propagate the network
- learning functions
- functions to manipulate patterns
- functions to load and save the network and pattern files
- functions for error treatment, search functions for names, functions to change default values etc.

The following paragraphs explains the interface functions in detail. All functions of this interface between the kernel and the user interface carry the prefix `krui_`...(kernel user interface functions).

Additionally there are some interface functions which are useful to build applications for ART networks. These functions carry the prefix `artui_`...(ART user interface functions).

15.2 Unit Functions

The following functions are available for manipulation of the cells and their components:

```
krui_getNoOfUnits()
krui_getNoOfSpecialUnits()
```

```
krui_getFirstUnit()
krui_getNextUnit()
krui_setCurrentUnit( int  UnitNo )
krui_getCurrentUnit()
krui_getUnitName( int  UnitNo )
krui_setUnitName( int  UnitNo, char *unit_name )
krui_searchUnitName( char  *unit_name )
krui_searchNextUnitName( void )
krui_getNoOfTTypeUnits()
krui_getUnitOutFuncName( int  UnitNo )
krui_setUnitOutFunc( int  UnitNo, char *unitOutFuncName )
krui_getUnitActFuncName( int  UnitNo )
krui_setUnitActFunc( int  UnitNo, char *unitActFuncName )
krui_getUnitFTypeName( int UnitNo )
krui_getUnitActivation( int  UnitNo )
krui_setUnitActivation( int  UnitNo, FlintType  unit_activation )
krui_getUnitInitialActivation( int  UnitNo )
krui_setUnitInitialActivation( int  UnitNo, FlintType unit_i_activation)
krui_getUnitOutput( int  UnitNo )
krui_setUnitOutput( int  UnitNo, FlintType  unit_output )
krui_getUnitBias( int  UnitNo )
krui_setUnitBias( int  UnitNo, FlintType  unit_bias )
krui_getUnitSubnetNo( int UnitNo )
krui_setUnitSubnetNo( int UnitNo, int  subnet_no)
krui_getUnitLayerNo( int UnitNo )
krui_setUnitLayerNo( int UnitNo, unsigned short layer_bitField )
krui_getUnitPosition( int  UnitNo, struct PosType  *position )
krui_setUnitPosition( int  UnitNo, struct PosType  *position )
krui_getUnitNoAtPosition( struct PosType  *position, int subnet_no )
krui_getUnitNoNearPosition( struct PosType  *position, int subnet_no,
                            int range, int gridWidth )
krui_getXYTransTable( struct TransTable  * *xy_trans_tbl_ptr )
krui_getUnitCenters( int unit_no, int center_no,
                     struct PositionVector  * *unit_center )
krui_setUnitCenters( int unit_no, int center_no,
                     struct PositionVector  *unit_center )
krui_getUnitTType( int  UnitNo )
krui_setUnitTType( int  UnitNo, int  UnitTType )
krui_freezeUnit( int  UnitNo )
krui_unfreezeUnit( int  UnitNo )
krui_isUnitFrozen( int  UnitNo )
krui_getUnitInputType( UnitNo )
krui_getUnitValueA(int UnitNo)
krui_setUnitValueA(int UnitNo, FlintTypeParam unit_valueA)
krui_createDefaultUnit()
krui_createUnit( char  *unit_name, char  *out_func_name,
                 char  *act_func_name, FlintType  act,
```

```
                 FlintType  i_act, FlintType  out,
                 FlintType  bias)
krui_createFTypeUnit( char  *FType_name)
krui_setUnitFType( int  UnitNo, char *FTypeName )
krui_copyUnit( int  UnitNo, int copy_mode )
krui_deleteUnitList( int no_of_units, int unit_list[] )
```

15.2.1 Unit Enquiry and Manipulation Functions

`int  krui_getNoOfUnits()`
determines the number of units in the neural net.

`int  krui_getNoOfSpecialUnits()`
determines the number of special units in the neural net.

`int  krui_getFirstUnit()`
Many interface functions refer to a current unit or site. `krui_getFirstUnit()` selects the (chronological) first unit of the network and makes it current. If this unit has sites, the chronological first site becomes current. The function returns 0 if no units are defined.

`int  krui_getNextUnit()`
selects the next unit in the net, as well as its first site (if present); returns 0 if no more units exist.

`krui_err  krui_setCurrentUnit( int  UnitNo )`
makes the unit with number *UnitNo* current unit; returns an error code if no unit with the specified number exists.

`int  krui_getCurrentUnit()`
determines the number of the current unit (0 if not defined)

```
char     *krui_getUnitName( int  UnitNo )
krui_err  krui_setUnitName( int  UnitNo, char *unit_name )
```
determines/sets the name of the unit. `krui_setUnitName` returns NULL if no unit with the specified number exists.

`int  krui_searchUnitName( char  *unit_name )`
searches for a unit with the given name. Returns the first unit number if a unit with the given name was found, 0 otherwise.

`int  krui_searchNextUnitName( void )`
searches for the next unit with the given name. Returns the next unit number if a unit with the given name was found, 0 otherwise. `krui_searchUnitName( unit_name )` has to be called before at least once, to confirm the unit name. Returns error code if no units are defined.

```
char  *krui_getUnitOutFuncName( int  UnitNo )
char  *krui_getUnitActFuncName( int  UnitNo )
```
determines the output function resp. activation function of the unit.

```
krui_err  krui_setUnitOutFunc( int  UnitNo, char *unitOutFuncName )
krui_err  krui_setUnitActFunc( int  UnitNo, char *unitActFuncName )
```
sets the output function resp. activation function of the unit. Returns an error code if the function name is unknown, i.e. if the name does not appear in the function table as output or activation function. The f-type of the unit is deleted.

```
char  *krui_getUnitFTypeName( int UnitNo )
```
yields the f-type of the unit; returns NULL if the unit has no prototype.

```
FlintType  krui_getUnitActivation( int UnitNo )
krui_err   krui_setUnitActivation( int UnitNo,
                                   FlintTypeParam unit_activation )
```
returns/sets the activation of the unit.

```
FlintType krui_getUnitInitialActivation( int  UnitNo )
void      krui_setUnitInitialActivation( int  UnitNo,
                                         FlintType unit_i_activation )
```
returns/sets the initial activation of the unit, i.e. the activation after loading the net. See also `krui_resetNet()`.

```
FlintType  krui_getUnitOutput( int  UnitNo )
krui_err   krui_setUnitOutput( int  unit_no, FlintTypeParam  unit_output )
```
returns/sets the output value of the unit.

```
FlintType  krui_getUnitBias( int  UnitNo )
void  krui_setUnitBias( int  UnitNo, FlintType  unit_bias )
```
returns/sets the bias (threshold) of the unit.

```
int   krui_getUnitSubnetNo( int UnitNo )
void  krui_setUnitSubnetNo( int UnitNo, int  subnet_no)
```
returns/sets the subnet number of the unit (the range of subnet numbers is -32736 to +32735).

```
unsigned short  krui_getUnitLayerNo( int UnitNo )
void            krui_setUnitLayerNo( int UnitNo,int layer_no)
```
returns/sets the layer number (16 Bit integer).

```
void  krui_getUnitPosition( int  UnitNo, struct PosType  *position )
void  krui_setUnitPosition( int  UnitNo, struct PosType  *position )
```
determines/sets the (graphical) position of the unit. See also include file `glob_typ.h` for the definition of PosType.

```
int  krui_getUnitNoAtPosition( struct PosType  *position, int subnet_no )
```
yields the unit number of a unit with the given position and subnet number; returns 0 if no such unit exists.

```
int  krui_getUnitNoNearPosition( struct PosType  *position,
                                 int subnet_no, int range,
                                 int gridWidth )
```
yields a unit in the surrounding (defined by *range*) of the given position with the given graphic resolution *gridWidth*; otherwise like `krui_getUnitNoAtPosition(...)`.

```
krui_err krui_getUnitCenters( int unit_no, int center_no,
                              struct PositionVector  * *unit_center )
```
returns the 3D-transformation center of the specified unit and center number. Function has no effect on the current unit. Returns error number if unit or center number is invalid or if the SNNS-kernel isn't a 3D-kernel.

```
krui_err krui_setUnitCenters( int unit_no, int center_no,
                              struct PositionVector  *unit_center )
```
sets the 3D-transformation center and center number of the specified unit. Function has no effect on the current unit. Returns error number if unit or center number is invalid or if the SNNS-kernel isn't a 3D-kernel.

```
krui_err krui_getXYTransTable( dummy )
```
returns the base address of the XY-translation table. Returns error code if the SNNS-kernel isn't a 3D-kernel.

```
int       krui_getUnitTType( int  UnitNo )
krui_err  krui_setUnitTType( int  UnitNo, int  UnitTType )
```
gets/sets the IO-type[1] (i.e. input, output, hidden) of the unit. See include file `glob_typ.h` for IO-type constants. `Set` yields an error code if the IO-type is invalid.

```
krui_err  krui_freezeUnit( int  unit_no )
```
freezes the output and the activation value of the unit, i.e. these values are not updated anymore.

```
krui_err  krui_unfreezeUnit( int  unit_no )
```
switches the computation of output and activation values on again.

```
bool  krui_isUnitFrozen( int  unit_no )
```
yields TRUE if the unit is frozen, else FALSE.

```
int  krui_getUnitInputType( UnitNo )
```
yields the input type. There are three kinds of input types:

`NO_INPUTS`: the unit doesn't have inputs (yet).

`SITES`: the unit has one or more sites (and therefore no direct inputs).

`DIRECT_LINKS`: the unit has direct inputs (and no sites).

See also file `glob_typ.h`.

```
FlintType krui_getUnitValueA(int UnitNo)
void      krui_setUnitValueA(int UnitNo, FlintTypeParam unit_valueA)
```
returns and sets the Value A field of the unit structure.

[1]The term T-type was changed to IO-type after completion of the kernel

15.2.2 Unit Definition Functions

`int krui_createDefaultUnit()`
creates a unit with the properties of the (definable) default values of the kernel. The default unit has the following properties:

- standard activation and output function
- standard activation and bias
- standard position-, subnet-, and layer number
- default IO type
- no unit prototype
- no sites
- no inputs or outputs
- no unit name

Returns the number of the new unit or a (negative) error code. See also include file `kr_def.h`.

```
int  krui_createUnit( char  *unit_name, char  *out_func_name,
                      char  *act_func_name, FlintTypeParam i_act,
                      FlintTypeParam  bias)
```
creates a unit with selectable properties; otherwise like `krui_createDefaultUnit()`. There are the following defaults:

- standard position-, subnet-, and layer number
- default IO type
- no unit prototype
- no sites
- no inputs or outputs

Returns the number of the new unit or a (negative) error code. See also include file `kr_def.h`.

`int krui_createFTypeUnit( char *FType_name)`
creates a unit with the properties of the (previously defined) prototype. It has the following default properties:

- standard position number, subnet number and layer number
- no inputs or outputs

The function returns the number of the new unit or a (negative) error code.

`krui_err krui_setUnitFType( int UnitNo, char *FTypeName )`
changes the structure of the unit to the intersection of the current type of the unit with the prototype; returns an error code if this operation has been failed.

`int krui_copyUnit( int UnitNo, int copy_mode)`
copies a unit according to the copy mode. Four different copy modes are available:

- copy unit with all input and output connections
- copy only input connections

- copy only output connections
- copy only the unit, no connections

Returns the number of the new unit or a (negative) error code. See `glob_typ.h` for reference of the definition of constants for the copy modes.

`krui_err krui_deleteUnitList( int no_of_units, int unit_list[] )`
deletes 'no_of_units' from the network. The numbers of the units that have to be deleted are listed up in an array of integers beginning with index 0. This array is passed to parameter 'unit_list'. Removes all links to and from these units.

15.3 Site Functions

Before input functions (sites) can be set for units, they first have to be defined. To define it, each site is assigned a name by the user. Sites can be selected by using this name. For the definition of sites, the following functions are available:

```
krui_createSiteTableEntry( char  *site_name, char  *site_func )
krui_changeSiteTableEntry( char  *old_site_name, char  *new_site_name,
                           char  *new_site_func )
krui_deleteSiteTableEntry( char *site_name )
krui_getFirstSiteTableEntry( char * *site_name, char * *site_func )
krui_getNextSiteTableEntry( char * *site_name, char * *site_func )
krui_getSiteTableFuncName( char  *site_name )
krui_setFirstSite( void )
krui_setNextSite( void )
krui_setSite( char  *site_name )
krui_getSiteValue()
krui_getSiteName()
krui_setSiteName( char  *site_name )
krui_getSiteFuncName()
krui_addSite( char *site_name )
krui_deleteSite()
```

15.3.1 Functions for the Definition of Sites

`krui_err  krui_createSiteTableEntry( char  *site_name, char  *site_func )`
defines the correspondence between site function and name of the site. Error codes are generated for site names already used, invalid site functions, or problems with the memory allocation.

```
krui_err  krui_changeSiteTableEntry( char  *old_site_name,
                                     char  *new_site_name,
                                     char  *new_site_func )
```
changes the correspondence between site function and name of the site. All sites in the network with the name `old_site_name` change their name and function. Error codes are

generated for already defined site names, invalid new site function, or problems with the memory allocation.

`krui_err  krui_deleteSiteTableEntry( char *site_name )`
deletes a site in the site table. This is possible only if there exist no sites in the network with that name. Returns an error code if there are still sites with this name in the net.

`bool  krui_getFirstSiteTableEntry( char * *site_name, char * *site_func )`
`bool  krui_getNextSiteTableEntry ( char * *site_name, char * *site_func )`
returns the first/next pair of site name and site function. The return code is TRUE if there is (still) an entry in the site table, else FALSE.

`char  *krui_getSiteTableFuncName( char  *site_name )`
returns the name of the site function assigned to the site. If no site with this name exists, a pointer to NULL is returned.

15.3.2 Functions for the Manipulation of Sites

`bool  krui_setFirstSite( void )`
initializes the first site of the current unit, i.e. the first site of the current unit becomes current site. If the current unit doesn't have sites, FALSE is returned, else TRUE.

`bool  krui_setNextSite( void )`
initializes the next site of the current unit. If the unit doesn't have more sites, FALSE is returned.

`krui_err  krui_setSite( char  *site_name )`
initializes the given site of the current unit. An error code is generated if the unit doesn't have sites, the site name is invalid, or the unit doesn't have a site with that name.

`FlintType  krui_getSiteValue()`
`char      *krui_getSiteFuncName()`
returns the name/value of the site function of the current site.

`char  *krui_getSiteName()`
returns the name of the current site.

`krui_err  krui_setSiteName( char  *site_name )`
changes the name (and thereby also the site function) of the current site. An error code is returned if the site name is unknown. The f-type of the unit is erased.

`krui_err krui_addSite( char *site_name )`
adds a new site to the current unit. The new site is inserted in front, i.e. it becomes the first site of the unit. Therefore it is possible to make the new site current by a call to `krui_setFirstSite()`. `krui_addSite(...)` has no effect on the current site! Error codes are generated if the unit has direct input connections, the site name is invalid, or problems with the memory allocation occurred. The functionality type of the unit will be cleared.

`bool krui_deleteSite()`
deletes the current site of the current unit and all input connections to that site. The functionality type of the unit is also erased. `krui_setFirstSite()` or `krui_setNextSite()` has to be called before at least once, to confirm the current site/unit. After the deletion the next available site becomes current. The return code is TRUE if further sites exist, else FALSE. The following program is sufficient to delete all sites of a unit:

```
if ( krui_setFirstSite() )
  while ( krui_deleteSite() ) { }
```

15.4 Link Functions

The following functions are available to define or determine the topology of the network:

```
krui_getFirstPredUnit( FlintType  *strength )
krui_getNextPredUnit( FlintType   *strength )
krui_getCurrentPredUnit( FlintType  *strength )
krui_getFirstSuccUnit( int  UnitNo, FlintType  *strength )
krui_getNextSuccUnit( FlintType  *strength )
krui_isConnected( int  source_unit_no )
krui_areConnected( int source_unit_no, int target_unit_no,
                   FlintType  *weight )
krui_getLinkWeight()
krui_setLinkWeight( FlintTypeParam  weight )
krui_createLink( int source_unit_no, FlintTypeParam  weight )
krui_deleteLink()
krui_deleteAllInputLinks()
krui_deleteAllOutputLinks()
krui_jogWeights( FlintTypeParam  minus, FlintTypeParam  plus)
```

`int krui_getFirstPredUnit( FlintType *strength )`
determines the unit number of the predecessor unit of the current unit and site; returns 0 if no such unit exists, i.e. if the current unit has no inputs. If a predecessor unit exists, the connection between the two units becomes current and its strength is returned.

`int krui_getNextPredUnit( FlintType *strength )`
gets another predecessor unit of the current unit/site (returns 0 if no more exist). Otherwise like `krui_getFirstPredUnit(...)`.

`int krui_getCurrentPredUnit( FlintType *strength )`
yields the current predecessor unit.

`int krui_getFirstSuccUnit( int UnitNo, FlintType *strength )`
yields unit number and connection strength of the first successor unit to the current unit. The return code is 0 if no such unit exists, i.e. the current unit has no outputs. If a successor unit exists, the connection between the two units becomes current. If the successor unit has sites, the site connected with this link becomes current site. The function is slow, because

the units are connected only backwards (lookup time is proportional to the number of connections in the net).

`int krui_getNextSuccUnit( FlintType *strength )`
gets another successor unit of the current unit (returns 0 if no other successors exist). Otherwise like `krui_getFirstSuccUnit(...)`. The function is slow, because the units are only connected backwards.

`bool krui_isConnected( int source_unit_no )`
checks, whether there is a connection between the current unit and the source unit. If this is true, this link becomes current and TRUE is returned.

```
bool  krui_areConnected( int source_unit_no, int target_unit_no,
                         FlintType  *weight )
```
checks, whether there is a connection between the source unit and the target unit. In contrast to `krui_isConnected(...)` this function traverses sites during the search. If there is such a connection, this link becomes current and TRUE is returned.

```
FlintType  krui_getLinkWeight()
void       krui_setLinkWeight( FlintTypeParam  weight )
```
determines/sets the connection weight of the current link.

`krui_err krui_createLink( int source_unit_no, FlintTypeParam weight )`
creates a new link between the current unit/site and the source unit. An error code is generated if a link between these two units already exists, or if the source unit does not exist.

`krui_err krui_deleteLink()`
deletes the current link. To delete a connection between the current unit/site and the source unit a sequence of `krui_isConnected( source_unit_no )` and `krui_deleteLink()` is ideal.

```
krui_err  krui_deleteAllInputLinks()
krui_err  krui_deleteAllOutputLinks()
```
deletes all inputs/outputs at the current unit/site.

`void krui_jogWeights( FlintTypeParam minus, FlintTypeParam plus)`
adds uniform distributed random values to the connection weights of the network. Minus must be less then plus. See also `krui_setSeedNo(...)`.

15.5 Functions for the Manipulation of Prototypes

By describing the characteristic properties of units, like activation/output function and sites, the user can define unit prototypes (called f-types in SNNS). Thereby the user can create a library of units. It is a big advantage that each change to the prototypes in the library affects all units of this f-type in the whole network. This means, that all units of a certain type are updated with a change in the library. With the following functions prototypes can be defined and manipulated:

```
krui_setFirstFTypeEntry()
krui_setNextFTypeEntry()
krui_setFTypeEntry( char  *Ftype_symbol )
krui_getFTypeName()
krui_setFTypeName( char  *unitFType_name )
krui_getFTypeActFuncName()
krui_setFTypeActFunc( char  *act_func_name )
krui_getFTypeOutFuncName()
krui_setFTypeOutFunc( char  *out_func_name )
krui_setFirstFTypeSite()
krui_setNextFTypeSite()
krui_getFTypeSiteName()
krui_setFTypeSiteName( char  *FType_site_name )
krui_createFTypeEntry( char *FType_symbol,char *act_func,
                       char *out_func, int no_of_sites,
                       char * *array_of_site_names )
krui_deleteFTypeEntry( char *  FType_symbol )
```

```
bool  krui_setFirstFTypeEntry()
bool  krui_setNextFTypeEntry()
```
initializes the first/next prototype and makes it current. The return code is FALSE if no unit types are defined, otherwise TRUE.

```
bool  krui_setFTypeEntry( char *Ftype_symbol )
```
selects a prototype by a name and returns TRUE if the name exists.

```
char  *krui_getFTypeName()
```
determines the name of the current prototype.

```
krui_err  krui_setFTypeName( char  *unitFType_name )
```
changes the name of the current prototype. The name has to be unambiguous, i.e. all names have to be different. If the name is ambiguous, or if memory allocation failed, an error code is returned.

```
char  *krui_getFTypeActFuncName()
```
determines the name of the activation function of the current prototype.

```
krui_err  krui_setFTypeActFunc( char * act_func_name )
```
changes the activation function of the current prototype; returns an error code if the given function is not a valid activation function. All units of the net that are derived from this prototype change their activation function.

```
char  *krui_getFTypeOutFuncName()
```
determines the name of the output function of the current prototype.

`krui_err krui_setFTypeOutFunc( char *out_func_name )`
changes the output function of the current prototype; returns an error code if the given function is not a valid output function. All units of the net that are derived from this prototype change their output function.

`bool krui_setFirstFTypeSite()`
selects the first site of the prototype (this site becomes current prototype site); returns TRUE if the prototype has sites.

`bool krui_setNextFTypeSite()`
selects the next site of the prototype (this site becomes current prototype site); returns TRUE if the prototype has more sites.

`char *krui_getFTypeSiteName()`
determines the name of the current prototype site.

`krui_err krui_setFTypeSiteName( char *FType_site_name )`
changes the name (and also the site function) of the current prototype site. All units of the net that are derived from this prototype change their site names or site functions. An error code is generated if the new site name is not yet defined.

```
krui_err krui_createFTypeEntry( char *FType_symbol,char *act_func,
                                char *out_func, int no_of_sites,
                                char * *array_of_site_names )
```
defines the prototype of a unit. If the prototype is supposed to have sites, i.e. `no_of_sites > 0`, an array of pointers to strings is needed to define the sites. These pointers must point to strings containing the names of the sites. The number of pointers in the arrays must match `no_of_sites`. An error code is generated if the site names are ambiguous, one or several site names are unknown, the prototype name is ambiguous, or a memory allocation error occurred.

`void krui_deleteFTypeEntry( char *FType_symbol )`
deletes the specified prototype. If there are still units in the net which are derived from this prototype, they loose their unit type.

15.6 Functions to Read the Function Table

`int krui_getNoOfFunctions()`
determines the number of available functions (unit, site, and learning functions).

```
void krui_getFuncInfo( int  func_no,
                       char * *func_name,
                       int  *func_type )
```
determines the name and type (activation, output, site, or learning function) from the function table. See include file `glob_typ.h` for the definition of the function types.

`bool krui_isFunction( char *func_name, int func_type )`
returns TRUE if the specified function is a valid function.

```
bool krui_getFuncParamInfo( char *func_name, int  func_type,
                            int *no_of_input_params,
                            int *no_of_output_params )
```
returns the number of input and output parameters of the given learning, update or initialization function. Returns TRUE if the given function exists, FALSE otherwise.

15.7 Network Initialization Functions

`krui_err  krui_setInitialisationFunc( char  *init_func )`
changes the initialization function; returns an error code if the initialization function is unknown.

`char  *krui_getInitialisationFunc( void )`
returns the current initialization function. The default initialization function is *'Randomize_Weights'* (see also `kr_def.h`).

`krui_err  krui_initializeNet( float *parameterArray, int  NoOfParams )`
initializes the network with the current initialization function.

15.8 Functions for Activation Propagation in the Network

`krui_err  krui_updateSingleUnit( int  UnitNo )`
evaluates the net input, the activation and the output value of the specified unit; returns an error code if the unit doesn't exist. `krui_updateSingleUnit(...)` also evaluates 'frozen' units.

`char  *krui_getUpdateFunc( void )`
returns the current update function. The default update function is *'Serial_Order'* (see also `kr_def.h`).

`krui_err  krui_setUpdateFunc( char  *update_func )`
Changes the current update function; returns an error code if the update function is invalid.

`krui_err  krui_updateNet( float *parameterArray, int  NoOfParams )`
updates the network according to the update function. The network should be a feed-forward type if one wants to update the network with the topological update function, otherwise the function returns a warning message. To propagate a pattern through the network the use of the following function calls is recommended:

```
krui_setPatternNo( pat_no );
krui_showPattern( OUTPUT_NOTHING );
krui_updateNet( parameterArray, NoOfParams );
```

See also `krui_setSeedNo` for initializing the pseudo random number generator. The function returns an error code if an error occurred. The following update functions are available:

- synchronous firing: the units of the network all change their activation at the same time.
- chronological order: the user defines the order in which the units change their activations.
- random order: the activations are computed in random order. It may happen that some units are updated several times while others are not updated at all.
- random permutation: the activations of all units are computed exactly once, but in a random order.
- topological order: the units change their activations according to their topological order. This mode should be selected only with nets that are free of cycles (feed-forward nets).

The topological order propagation method computes the stable activation pattern of the net in just one cycle. It is therefore the method of choice in cycle free nets. In other modes, depending upon the number of layers in the network, several cycles are required to reach a stable activation pattern (if this is possible at all).

15.9 Learning and Pruning Functions

```
krui_err  krui_setLearnFunc( char *learning_function)
krui_err  krui_setFFLearnFunc( char *FF_learning_function)
krui_err  krui_setPrunnFunc( char *pruning_function)
```
selects the learning / subordinate learning / pruning function; returns an error code if the given function is unknown.

```
char  *krui_getLearnFunc( void )
char  *krui_getFFLearnFunc( void )
char  *krui_getPrunFunc( void )
```
returns the name of the current learning / subordinate learning / pruning function. The default learning / subordinate learning function is *'Std_Backpropagation'*, the default pruning function is *'MagPruning'* (see also `kr_def.h`).

```
krui_err  krui_learnAllPatterns( float *parameterInArray,
                                 int  NoOfInParams,
                                 float  * *parameterOutArray,
                                 int *NoOfOutParams )
krui_err  krui_learnAllPatternsFF( float *parameterInArray,
                                   int  NoOfInParams,
                                   float  * *parameterOutArray,
                                   int *NoOfOutParams )
```
learns all patterns (each consisting of an input/output pair) using the current learning /

subordinate learning function. `parameterInArray` contains the learning parameter(s). `NoOfInParams` stores the number of learning parameters. `parameterOutArray` returns the results of the learning function. This array is a static array defined in the learning function. `*NoOfOutParams` points to an integer value that contains the number of output parameters from the current learning function. The function returns an error code if memory allocation has failed or if the learning has failed. Patterns must be loaded before calling this function.

```
krui_err  krui_learnSinglePattern( int  pattern_no,
                                   float   *parameterInArray,
                                   int NoOfInParams,
                                   float  * *parameterOutArray,
                                   int *NoOfOutParams )
krui_err  krui_learnSinglePatternFF( int  pattern_no,
                                     float   *parameterInArray,
                                     int NoOfInParams,
                                     float  * *parameterOutArray,
                                     int *NoOfOutParams )
```

same as `krui_learnAllPatterns( ... )` or `krui_learnAllPatternsFF( ... )` respectively, but teaches only the current pattern.

15.10 Functions for the Manipulation of Patterns

`krui_err  krui_setPatternNo( int  patter_no )`
sets the current pattern; returns an error code if the pattern number is invalid.

`krui_err  krui_deletePattern( void )`
deletes the current pattern.

`krui_err  krui_modifyPattern( void )`
modifies the current pattern. Sets the pattern to the current activation of the units.

`krui_err  krui_showPattern( int  mode )`
outputs a pattern on the activation or output values of the input/output units. The following modes are possible:

- `OUTPUT_NOTHING`: stores the input pattern in the activation of the input units.
- `OUTPUT_ACT`: like `OUTPUT_NOTHING`, but stores also the output pattern in the activation of the output units.
- `OUTPUT_OUT`: like `OUTPUT_ACT`, additionally a new output value of the output units is computed.

`krui_showPattern(...)`
draws pattern on the display. Generates an error code if the number of input and output units does not correspond with the previously loaded pattern. The constants of the various modes are defined in `glob_typ.h`.

`krui_err krui_newPattern( void )`
creates a new pattern (an input/output pair). A pattern can be created by modifying the activation value of the input/output units. The function returns an error code if there is insufficient memory or the number of input/output units is incompatible with the pattern.

Note: `krui_newPattern()` switches pattern shuffling off. For shuffling the new patterns call:

```
krui_newPattern(...)
krui_shufflePatterns( TRUE )
```

`void krui_deleteAllPatterns()`
deletes all previously defined patterns in main memory.

`krui_err krui_shufflePatterns( bool on_or_off )`
shuffles the order of the patterns if `on_or_off` is true. If `on_or_off` is false the original order can be restored. See also `krui_setSeedNo(...)`.

`krui_err krui_shuffleSubPatterns(bool on_or_off)`
shuffles sub pattern pairs by using pseudo random generator.
`krui_shuffleSubPatterns(TRUE)` switches shuffeling of sub patterns on,
`krui_shuffleSubPatterns(FALSE)` switches shuffeling of sub patterns off.
The default presetting is `krui_shuffleSubPatterns (FALSE)`.

`int krui_getNoOfPatterns( void )`
returns the actual number of patterns (0 if no patterns have been loaded).

`krui_err krui_allocNewPatternSet(int *set_no)`
returns the number of the allocated pattern set. In case of an error the error code will be returned.

`krui_err krui_setCurrPatSet(int number)`
chooses the number of the current pattern set. The number ranges from 0 to n-1.

`krui_err krui_deletePatSet(int number)`
deletes all patterns of the pattern set with the given number.

```
krui_err  krui_GetPatInfo(pattern_set_info *set_info,
                          pattern_descriptor *pat_info)
```
gets all available information concerning the current pattern set and the current pattern.

```
krui_err  kruiDefShowSubPat(int *insize,int *outsize,int,
                            int *inpos,int *outpos)
```
defines the sub pattern that will be shown with the next call of `krui_showPattern`.

```
krui_err  krui_DefTrainSubPat(int *insize,int *outsize,int *instep,
                              int *outstep, int *max_n_pos)
```
defines how sub pattern should be generated during training. `krui_DefTrainSubPat()` has to be called before any training can take place.

```
krui_err   krui_AlignSubPat(int *inpos,int *outpos,int *no)
```
aligns the given sub pattern position to a valid position which fits the defined sub pattern training scheme (`krui_DefTrainSubPat`).

```
krui_err  krui_GetShapeOfSubPattern(int *insize,int *outsize,int *inpos,
                                    int *outpos,int *n_pos)
```
gets the shape of the sub pattern by using the current set,current pattern and the current train scheme (defined with `krui_DefTrainSubPat`).

15.11 File I/O Functions

```
krui_err  krui_loadNet( char  *filename, char  * *netname )
krui_err  krui_saveNet( char  *filename, char  *netname)
```
loads/saves a network from/to disk and generates an internal network structure.

```
krui_err krui_loadNewPatterns(char *filename,int *set_no)
krui_err  krui_saveNewPatterns(char *filename,int set_no)
```
saves and loads a pattern file with new style conventions.
The pattern set with the given number will be saved and loaded.

```
krui_err krui_saveResultParam(char *filename,bool create,int startpattern,
                      int endpattern,bool includeinput,bool includeoutput,
                      float *Update_param_array, int NoOfUpdateParam)
```
Saves the network result which depends on the loaded patterns. If create is false,the new file will be appended to an existing file. Startpattern and endpattern determines the range of patterns to use. The input patterns and the teaching output patterns can be included.

An error code is returned if an error occurred during the execution of the I/O functions.

15.12 Functions to Search the Symbol Table

```
bool  krui_getFirstSymbolTableEntry( char * *symbol_name,
                                    int *symbol_type )
bool  krui_getNextSymbolTableEntry( char * *symbol_name,
                                    int *symbol_type )
```
determines the name and type of the first/next entry in the symbol table and returns TRUE if another entry exists.

```
bool  krui_symbolSearch( char *symbol, int  symbol_type)
```
returns TRUE if the given symbol exists in the symbol table.

15.13 Miscelaneous other Interface Functions

```
char  *krui_getVersion()
```
determines the version number of the SNNS kernel.

```
void  krui_getNetInfo( int  *no_of_sites, int  *no_of_links,
                       int  *no_of_STable_entries,
                       int  *no_of_FTable_entries )
```
gathers various information about the network.

```
void  krui_getUnitDefaults( FlintType  *act, FlintType  *bias,
                           int *io_type, int *subnet_no, int *layer_no,
                           char * *act_func, char * *out_func)
```
determines the default values for generating units. See also `krui_createDefaultUnit()` and `krui_createFTypeUnit(...)`.

```
krui_err  krui_setUnitDefaults( FlintTypeParam act, FlintTypeParam bias,
                               int io_type, int subnet_no, int layer_no,
                               char *act_func, char *out_func)
```
changes the default values; returns an error code if the IO-type or the activation/output function is unknown.

```
void  krui_setSeedNo( long seed )
```
initializes the random number generator. If `seed = 0`, the random number generator is re-initialized (this time really at random!).

```
int  krui_getNoOfInputUnits()
int  krui_getNoOfOutputUnits()
```
return the number of input/output units

```
int  krui_getNoOfSpecialInputUnits()
int  krui_getNoOfSpecialOutputUnits()
```
return the number of special input/output units

```
void  krui_resetNet()
```
sets the activation values of all units to their respective defaults.

15.14 Memory Management Functions

```
krui_err  krui_allocateUnits( int  number )
```
reserves `<number>` units in memory. Additional units can be requested by multiple calls to that function. This function doesn't have to be called, since the SNNS kernel always reserves enough memory for units, sites and links. If a large amount of units is needed, however, a call to `krui_allocateUnits(...)` eases the administration of system resources. If `krui_allocateUnits(...)` is never called, units are always requested in blocks of size `<UNIT_BLOCK>`. See also `kr_def.h`.

```
void krui_getMemoryManagerInfo(int *unit_bytes, int *site_bytes,
                               int *link_bytes, int *NTable_bytes,
                               int *STable_bytes, int *FTable_bytes)
```
determines the number of allocated (not the number of used) bytes for the various components.

`void  krui_deleteNet()`
deletes the network and frees the whole memory used for the representation of the data structures.

15.15 ART Interface Functions

The functions described in this paragraph are only useful if you use one of the ART models ART1, ART2 or ARTMAP. They are additional functions, not replacing any of the kernel interface functions described above. To use them, you have to include the files `art_ui.h` and `art_typ.h` in your application.

`krui_err artui_getClassifiedStatus( art_cl_status *status )`
returns the actual classification state of an ART network in parameter `status`. Type `art_cl_status` is described above. An SNNS error code is returned as function value. The function can be used for ART1, ART2 and ARTMAP models.

`krui_err artui_getClassNo( int *class_no )`
returns a number between 1 and M in parameter `class_no`, which is either the index of the actual winner unit in the F_2 layer of an ART1 or ART2 network or the one of the activated MAP unit in an ARTMAP network. It will return -1 as `class_no` if no actual class is active. An SNNS error code is returned as function value.

`krui_err  artui_getN (int *N )`
determines the number of F1-units in an ART1 or ART2 network.

`krui_err  artui_getM (int *M )`
determines the number of F2-units in an ART1 or ART2 network.

`krui_err  artui_getNa (int *Na )`
determines the number of F1a-units in an ARTMAP network.

`krui_err  artui_getMa (int *Ma )`
determines the number of F2a-units in an ARTMAP network

`krui_err  artui_getNb (int *Nb)`
determines the number of F1b-units in an ARTMAP network.

`krui_err  artui_getMb (int *Mb)`
determines the number of F2b-units in an ARTMAP network.

15.16 Error Messages of the Simulator Kernel

Most interface functions return an error code if the parameters contradict each other, or if an error occurred during execution. If no error occurred during execution of a kernel interface function, the function returns the code: KRERR_NO_ERROR. KRERR_NO_ERROR is equal to 0. The simulator kernel can generate 54 different error messages. The error code constants are defined in glob_typ.h. There is also a function to translate an error code into text:

```
char  *krui_error( int  error_code )
```

converts an error code to a string. The following error messages are used:

KRERR_NO_ERROR:	No Error
KRERR_INSUFFICIENT_MEM:	Insufficient memory
KRERR_UNIT_NO:	Invalid unit number
KRERR_OUTFUNC:	Invalid unit output function
KRERR_ACTFUNC:	Invalid unit activation function
KRERR_SITEFUNC:	Invalid site function
KRERR_CREATE_SITE:	Creation of sites is not permitted because unit has direct input links
KRERR_ALREADY_CONNECTED:	Creation of a link is not permitted because there exists already a link between these units
KRERR_CRITICAL_MALLOC:	allocation failed in critical operation.
KRERR_FTYPE_NAME:	Ftype name is not definite
KRERR_FTYPE_ENTRY:	Current Ftype entry is not defined
KRERR_COPYMODE:	Invalid copy mode
KRERR_NO_SITES:	Current unit does not have sites
KRERR_FROZEN:	Can't update unit because unit is frozen
KRERR_REDEF_SITE_NAME:	Redefinition of site name not permitted (site name already exists)
KRERR_UNDEF_SITE_NAME:	Site name is not defined
KRERR_NOT_3D:	Invalid Function: Not a 3D-Kernel
KRERR_DUPLICATED_SITE:	Unit has already a site with this name
KRERR_INUSE_SITE:	Can not delete site table entry because site is in use
KRERR_FTYPE_SITE:	Current Ftype site is not defined
KRERR_FTYPE_SYMBOL:	Given symbol is not defined in the symbol table
KRERR_IO:	Physical I/O error
KRERR_SAVE_LINE_LEN:	Creation of output file failed (line length, limit exceeded)
KRERR_NET_DEPTH:	The depth of the network does not fit the learning function
KRERR_NO_UNITS:	No Units defined
KRERR_EOF:	Unexpected EOF
KRERR_LINE_LENGTH:	Line length exceeded

KRERR_FILE_FORMAT:	Incompatible file format
KRERR_FILE_OPEN:	Can not open file
KRERR_FILE_SYNTAX:	Syntax error at line
KRERR_MALLOC1:	Memory allocation error 1
KRERR_TTYPE:	Topologic type invalid
KRERR_SYMBOL:	Symbol pattern invalid (must match [A-Za-z]^[|,]*)
KRERR_NO_SUCH_SITE:	Current unit does not have a site with this name
KRERR_NO_HIDDEN_UNITS:	No hidden units defined
KRERR_CYCLES:	cycle(s):
KRERR_DEAD_UNITS:	dead unit(s):
KRERR_INPUT_PATTERNS:	Pattern file contains not the same number of input units as the network
KRERR_OUTPUT_PATTERNS:	Pattern file contains not the same number of output units as the network
KRERR_CHANGED_I_UNITS:	Number of input units has changed
KRERR_CHANGED_O_UNITS:	Number of output units has changed
KRERR_NO_INPUT_UNITS:	No input units defined
KRERR_NO_OUTPUT_UNITS:	No output units defined
KRERR_NO_PATTERNS:	No patterns defined
KRERR_INCORE_PATTERNS:	Incore patterns incompatible with current network (remove loaded patterns before loading network)
KRERR_PATTERN_NO:	Invalid pattern number
KRERR_LEARNING_FUNC:	Invalid learning function
KRERR_PARAMETERS:	Invalid parameters
KRERR_UPDATE_FUNC:	Invalid update function
KRERR_INIT_FUNC:	Invalid initialization function
KRERR_DERIV_FUNC:	Derivation function of the activation function does not exist
KRERR_I_UNITS_CONNECT:	input unit(s) with input connections to other units
KRERR_O_UNITS_CONNECT:	output unit(s) with output connections to other units
KRERR_TOPOMODE:	Invalid topological sorting mode
KRERR_LEARNING_SITES:	Learning function doesn't support sites
KRERR_SITES_NO_SUPPORT:	Sites are not supported
KRERR_NO_MASPAR_KERNEL:	This isn't a MasPar Kernel
KRERR_NOT_NEIGHBOUR_LAYER:	Connection(s) between unit(s) in non-neighbour layers are not supported
KRERR_MUCH_LAYERS:	The number of layers is too high
KRERR_NOT_FULLY_CONNECTED:	The network layers aren't fully connected
KRERR_MODE_FF1_INVALID_OP:	This operation is not allowed in parallel kernel mode
KRERR_NET_TRANSFORM:	Change of network type isn't possible in parallel kernel mode

KRERR_NO_CURRENT_LINK:	No current link defined
KRERR_NO_CURRENT_UNIT:	No current unit defined
KRERR_UNIT_NO_INPUT:	Current unit does't have any inputs
KRERR_TOPO_DEFINITION:	Invalid parameter in topologic definition section
KRERR_BAD_CONNECTION:	Creation of link between these units is not permitted
KRERR_MASPAR_NO_RESPOND:	MasPar doesn't respond
KRERR_NOT_IMPLEMENTED_YET:	This function isn't implemented yet
KRERR_NOT_PARRALLEL_MODE:	Kernel isn't in parallel mode
KRERR_MASPAR_OUT_OF_PE_MEM:	MasPar ran out of memory
KRERR_MASPAR_COMMUNICATE:	MasPar communication error
KRERR_MASPAR_OUT_OF_PE:	MasPar ran out of processors
KRERR_MISSING_DEFAULT_FUNC:	Missing default function (check function table)
KRERR_MASPAR_MULTI_OUT_FUNC:	MasPar kernel does not support multiple unit output functions
KRERR_MASPAR_MULTI_ACT_FUNC:	MasPar kernel doesn't support multiple unit activation functions
KRERR_NET_DEPTH:	The depth of the network does not fit to the learning function
KRERR_NO_OF_UNITS_IN_LAYER:	Wrong no. of units in layer
KRERR_UNIT_MISSING:	Unit missing or not correctly connected
KRERR_UNDETERMINED_UNIT:	Unit does not belong to a defined layer in the network
KRERR_ACT_FUNC:	Unit has wrong activation function
KRERR_OUT_FUNC:	Unit has wrong output function
KRERR_SITE_FUNC:	Unexpected site function at unit
KRERR_UNEXPECTED_SITES:	Unit is not expected to have sites
KRERR_UNEXPECTED_DIRECT_INPUTS:	Unit is expected to have sites
KRERR_SITE_MISSING:	Site missing at unit
KRERR_UNEXPECTED_LINK:	Unexpected link
KRERR_LINK_MISSING:	Missing link(s) to unit
KRERR_LINK_TO_WRONG_SITE:	Link ends at wrong site of target unit
KRERR_TOPOLOGY:	The network is not fitting the required topology
KRERR_PARAM_BETA:	Wrong beta parameter in unit bias value
KRERR_CC_ERROR1:	Topo_ptr_array is sorted in the wrong way
KRERR_CC_ERROR2:	There is no memory allocated
KRERR_CC_ERROR3:	Not enough memory to run Casscade
KRERR_CC_ERROR4:	Hidden unit with more than one recurent link detected
KRERR_CC_ERROR5:	Hidden unit without recurrent link found
KRERR_CC_ERROR6:	Hidden layer does not fit the required topology
KRERR_CC_ERROR7:	The first link of a hidden unit is not a recurent link

KRERR_CC_ERROR8:	Special unit with more than one recurent link detected
KRERR_CC_ERROR9:	First link of a special unit is not a recurent link
KRERR_CC_ERROR10:	Wrong update function
KRERR_CC_ERROR11:	Wrong init function
DLVQ_ERROR1:	There are empty classes
DLVQ_ERROR2:	There is a class lower than zero
DLVQ_ERROR3:	Wrong number of output units.
DLVQ_ERROR4:	This network is not fitting the required topology
DLVQ_ERROR5:	There isn't a unit for every class
KRERR_NP_NO_MORE_ENTRIES:	No more free pattern sets available
KRERR_NP_NO_SUCH_PATTERN_SET:	No such Pattern set defined
KRERR_NP_CURRENT_PATTERN:	No current pattern defined
KRERR_NP_DIMENSION:	Specified sub pattern does not fit into pattern
KRERR_NP_NO_SUCH_PATTERN:	No such pattern available
KRERR_NP_NO_CURRENT_PATTERN_SET:	No current pattern set defined
KRERR_NP_DOES_NOT_FIT:	Pattern (sub pattern) does not fit the network
KRERR_NP_NO_TRAIN_SCHEME:	No sub pattern shifting scheme defined
KRERR_NP_NO_OUTPUT_PATTERN:	Pattern contains no output information
KRERR_NP_INCOMPATIBLE_NEW:	New Pattern does not fit to existing set
KRERR_NP_PARAM_ZERO:	Illegal parameter value <= 0 specified
KRERR_IP_ISNOTINITED:	Paragon kernel is not initialized
KRERR_IP_ISINITED:	Paragon kernel is already initialized
KRERR_IP_SENDMSGFAILED:	Sending a message fails
KRERR_IP_MSGSYNTAX:	Syntax error in received message
KRERR_IP_UNKNOWENCMD:	Receive unknown command
KRERR_IP_LESSPAT:	There are less patterns than allocated nodes
KRERR_IP_WEIGHTUPDATE:	Weight update with global sum fails
KRERR_IP_LEARNFNOTPAR:	Learning function is not parallelized

Chapter 16

Transfer Functions

16.1 Predefined Transfer Functions

The following site, activation, and output functions are already predefined. Future releases of the kernel will have additional transfer functions.

Site Functions:

Function	**Formula**
`Linear`	$net_j(t) = \sum_i w_{ij} o_i$
`Produkt`	$net_j(t) = \prod_i w_{ij} o_i$
`PI`	$net_j(t) = \prod_i o_i$
`Max`	$net_j(t) = \max_i w_{ij} o_i$
`Min`	$net_j(t) = \min_i w_{ij} o_i$
`At_least_2`	$net_j(t) = \begin{cases} 0 & \text{if } \sum_i w_{ij} o_i < 2 \\ 1 & \text{else.} \end{cases}$

Several other site functions have been implemented for the ART models in SNNS: `At_least_2`, `At_least_1`, `At_most_0`, `Reciprocal`. These functions normally are not useful for other networks. So they are mentioned here, but not described in detail. For more information please refer to the section about the corresponding ART model in SNNS.

Activation functions:

Function	Formula
BAM	$a_j(t) = \begin{cases} 1 & \text{if } net_j(t) > 0 \\ a_j(t-1) & \text{if } net_j(t) = 0 \\ -1 & \text{if } net_j(t) < 0 \end{cases}$
BSB	$a_j(t) = net_j(t) * \theta_j$
Elliott	$a_j(t) = \frac{net_j(t)+\theta_j}{1+\lvert net_j(t)+\theta_j \rvert}$
Identity	$a_j(t) = net_j(t)$
IdentityPlusBias	$a_j(t) = net_j(t) + \theta_j$
Logistic	$a_j(t) = \frac{1}{1+e^{-(net_j(t)+\theta_j)}}$
Logistic_notInhibit	like Logistic, but skip input from units named "Inhibit"
Logistic_Tbl	like Logistic, but with table lookup instead of computation
MinOutPlusWeight	$a_j(t) = \min_i(w_{ij} + o_i)$
Perceptron	$a_j(t) = \begin{cases} 1 & \text{if } net_j(t) \geq \theta_j \\ 0 & \text{if } net_j(t) < \theta_j \end{cases}$
Product	$a_j(t) = \prod_i w_{ij} o_i$
RBF_Gaussian	see the chapter about RBFs in the user manual
RBF_MultiQuadratic	see the chapter about RBFs in the user manual
RBF_ThinPlateSpline	see the chapter about RBFs in the user manual
RM	$a_j(t) = \begin{cases} 0.85 * a_j(t-1)+ \\ 0.15 * net_j(t)* \\ (1 - a_j(t-1)) & \text{if } net_j(t) > 0 \\ 0.85 * a_j(t-1)+ \\ 0.15 * net_j(t)* \\ (1 + a_j(t-1)) & \text{if } net_j(t) \leq 0 \end{cases}$
Signum	$a_j(t) = \begin{cases} 1 & \text{if } net_j(t) > 0 \\ -1 & \text{if } net_j(t) \leq 0 \end{cases}$
Signum0	$a_j(t) = \begin{cases} 1 & \text{if } net_j(t) > 0 \\ 0 & \text{if } net_j(t) = 0 \\ -1 & \text{if } net_j(t) < 0 \end{cases}$
StepFunc	$a_j(t) = \begin{cases} 1 & \text{if } net_j(t) > 0 \\ 0 & \text{if } net_j(t) \leq 0 \end{cases}$
TanH	$a_j(t) = \tanh(net_j(t) + \theta_j)$
TanH_Xdiv2	$a_j(t) = \tanh(net_j(t)/2 + \theta_j)$

Several other activation functions have been implemented for the ART models in SNNS: `Less_than_0`, `At_most_0`, `At_least_2`, `At_least_1`, `Exactly_1`, `ART1_NC`, `ART2_Identity`, `ART2_Rec`, `ART2_NormP`, `ART2_NormV`, `ART2_NormW`, `ART2_NormIP`, `ART2_Rst`, `ARTMAP_NCa`, `ARTMAP_NCb`, `ARTMAP_DRho`.

These functions normally are not useful for other networks. So they are mentioned here, but not described in detail. For cascade correlation and time delay networks the following modified versions of regular activation functions have been implemented: `RCC_logistic`, `RCC_LogisticSym`, `RCC_Tanh`, `RCC_Linear`, `TD_Logistic`, `TD_Elliott`. They behave like the ordinary functions with the same name body.

Output Functions:

Function	**Formula**
`Identity`	$o_j(t) = a_j(t)$
`Clip_0_1`	$o_j(t) = \begin{cases} 0 & \text{if } a_j(t) \leq 0 \\ a_j(t) & \text{if } a_j(t) < 1 \\ 1 & \text{if } a_j(t) \geq 1 \end{cases}$
`Clip_1_1`	$o_j(t) = \begin{cases} -1 & \text{if } a_j(t) \leq -1 \\ a_j(t) & \text{if } -1 < a_j(t) < 1 \\ 1 & \text{if } a_j(t) \geq 1 \end{cases}$
`Threshold_0.5`	$o_j(t) = \begin{cases} 0 & \text{if } a_j(t) \leq 0.5 \\ 1 & \text{if } a_j(t) > 0.5 \end{cases}$

Two other output functions have been implemented for ART2 in SNNS: `ART2_Noise_PLin` and `ART2_Noise_ContDiff` These functions are only useful for the ART2 implementation So they are mentioned here, but not described in detail.

16.2 User Defined Transfer Functions

The group of transfer functions can be exented arbitrarily by the user. In order to make them available inside SNNS the following steps have to be performed:

1. In file "`.../SNNSv4.0/kernel/func_tbl.c` the name of the function has to be inserted in the function table. An example entry for an activation function would be
   "Act_MyFunction",ACT_FUNC, 0, 0, (FunctionPtr) MyFancyFunction,

   Notice that the second entry defines the type (activation, initialization, etc.) of the new function!

   If the new function is an activation function, the corresponding derivation function has also to be inserted in the function table. E.g.:
   "Act_MyFunction",ACT_DERIV_FUNC, 0, 0, (FunctionPtr)ACT_DERIV_MFF,

   This entry has to be given, even if no such derivation function exists in the mathematical sense. In that case `''ACT_DERIV_Dummy''` has to be specified as name of the derivation function.

   If the function exists, it has to be declared and implemented just as the activation function.

Please note, that activation and derivation function have to have the same name suffix (here: "MyFunction")!

2. The functions must be implemented as C programs in following files:

   activation functions in "`.../SNNSv4.0/kernel/trans_f.c`"
   output functions in "`.../SNNSv4.0/kernel/trans_f.c`"
   site functions in "`.../SNNSv4.0/kernel/trans_f.c`"
   initialization functions in "`.../SNNSv4.0/kernel/init_f.c`"
   learning functions in "`.../SNNSv4.0/kernel/learn_f.c`"
   update functions in "`.../SNNSv4.0/kernel/update_f.c`"

   The name of the implemented function has to match the name specified in the function table!

3. Function prototypes have to be defined in the corresponding header (.h) and private header files (.ph).

4. Go to the SNNSv4.0 root directory and build (using the command 'build') new executables. Remember that if you don't recompile, the modification to the kernel won't take any effect.

The new function should be available now in the user interface together with all predefined functions.

Appendix A

Kernel File Interface

A.1 The ASCII Network File Format

The ASCII representation of a network consists of the following parts:

- a header, which contains information about the net
- the definition of the teaching function
- the definition of the sites
- the definition of cell types
- the definition of the default values for cells
- the enumeration of the cells with all their characteristics
- the list of connections
- a list of subnet numbers
- a list of layer numbers

All parts, except the header and the enumeration of the cells, may be omitted. Each part may also be empty. It then consists only of the part title, the header line and the boarder marks (e.g.: `----|---|------`).

Entries in the *site definition section* do not contain any empty columns. The only empty column in the *type definition section* may be the *sites* column, in which case the cells of this type do not have sites.

Entries in the *unit definition section* have at least the columns *no.* (cell number) and *position* filled. The entries (rows) are sorted by increasing cell number. If column *typeName* is filled, the columns *act func*, *out func*, and *sites* remain empty.

Entries in the *connection definition section* have all columns filled. The respective cell does not have a site, if the column *site* is empty. The entries are sorted by increasing number of the target cell (column *target*). Each entry may have multiple entries in the column *sources*. In this case, the entries (number of the source cell and the connection strength) are separated by a comma and a blank, or by a comma and a newline (see example in the Appendix B).

The file may contain comment lines. Each line beginning with # is skipped by the SNNS-kernel.

A.2 Form of the Network File Entries

Columns are separated by the string ␣|␣. A row never exceeds 250 characters.

Strings may have arbitrary length. The compiler determines the length of each row containing strings (maximum string length + 2). Within the columns, the strings are stored left adjusted. Strings may not contain blanks, but all special characters except `|`. The first character of a string has to be a letter.

Integers may have an arbitrary number of digits. Cell numbers are always positive and not zero. Position coordinates may be positive or negative. The compiler determines the length of each row containing integers (maximum digit number + 2). Within the columns, the numbers are stored right adjusted.

Floats are always stored in fixed length with the format `Vx.yyyyy`, where **V** is the sign (+, - or blank), **x** is 0 or 1 and **y** is the rational part (5 digits behind the decimal point).

Rows containing floats are therefore always 10 characters long (8 + 1 blank on each side).

If a row contains several sites in the *type* or *unit definition section*, they are written below each other. They are separated in the following way: Directly after the first site follows a comma and a newline (`\n`). The next line starts with an arbitrary number of blanks or tabs in front of the next site.

The source of a connection is described by a pair, the cell number and the strength of the connection. It always has the format `nnn:Vx.yyyyy` with the following meaning:

- `nnn` Number of the source
- `Vx.yyyyy` Strength of the connection as a float value (format as described above)

The compiler determines the width of the column **nnn** by the highest cell number present. The cell numbers are written into the column right adjusted (according to the rules for integers). The column `Vx.yyyyy` has fixed width. Several source pairs in an entry to the *connection definition section* are separated by a comma and a blank. If the list of source pairs exceeds the length of one line, the line has to be parted after the following rule:

- Separation is always between pairs, never within them.
- The comma between pairs is always directly behind the last pair, i.e. remains in the old line.
- After a newline (`\n`) an arbitrary number of blanks or tabs may precede the next pair.

A.3 Grammar of the Network Files

A.3.1 Conventions

A.3.1.1 Lexical Elements of the Grammar

The lexical elements of the grammar which defines network files are listed as regular expresions. The first column lists the name of the symbol, the second the regular expresion defining it. The third column may contain comments.

- All terminals (characters) are put between `"..."`.
- Elements of sets are put between square brackets. Within the brackets, the characters represent themselves (even without `"`), and `-` defines a range of values. The class of digits is defined, e.g. as `["0"-"9"]`.
- Characters can be combined into groups with parenteses `()`.
- `x*` means, that the character or group x can occur zero or more times.
- `x+` means, that the character or group x must occur at least once, but may occur several times.
- `x?` means, that x can be omitted.
- `x{n}` means, that x has to occur exactly n times.
- `x|y` means, that either x or y has to occur.
- `*`, `+` and `{}` bind strongest, `?` is second, `|` binds weakest.
- Groups or classes of characters are treated like a single character with respect to priority.

A.3.1.2 Definition of the Grammar

The Grammar defining the interface is listed in a special form of EBNF.

- Parts between square brackets `[]` are facultative.
- `|` separates alternatives (like with terminal symbols).
- `{x}` means, that x may occur zero or more times.
- CSTRING is everything that is recognized as string by the C programming language.

A.3.2 Terminal Symbols

```
WHITESPACE                {" "|"\n"|"\t"}                     /* whitespaces */
BLANKS_TABS               {" "|"\t"}                          /* only blanks or tabs */
W_COL_SEP                 (" "|"\n"|"\t") {" "|"\n"|"\t"} "|" {" "|"\n"|"\t"}
                          /* at least one blank and the column separation */
COL_SEP                   {" "|"\n"|"\t"} "|" {" "|"\n"|"\t"}  /* column separation */
COMMA                     {" "|"\n"|"\t"} "," {" "|"\n"|"\t"}  /* at least the comma */
EOL                       {" "|"\n"|"\t"} "\n" {" "|"\n"|"\t"} /* at least "\n" */
CUT                       {" "|"\n"|"\t"} (" "|"\n"|"\t") {" "|"\n"|"\t"}
                          /* at least a blank, "\t", or "\n" */
COLON                     ":"


/* separation lines for different tables */

TWO_COLUMN_LINE           "-"+"|""-"+
THREE_COLUMN_LINE         "-"+"|""-"+"|""-"+
FOUR_COLUMN_LINE          "-"+"|""-"+"|""-"+"|""-"+
SIX_COLUMN_LINE           "-"+"|""-"+"|""-"+"|""-"+"|""-"+"|""-"+
SEVEN_COLUMN_LINE         "-"+"|""-"+"|""-"+"|""-"+"|""-"+"|""-"+"|""-"+
TEN_COLUMN_LINE           "-"+"|""-"+"|""-"+"|""-"+"|""-"+"|""-"+"|""-"+
                          "|""-"+"|""-"+"|""-"+

COMMENT                   {{" "|"\n"|"\t"} "#" CSTRING "\n" {" "|"\n"|"\t"}}

VERSION                   "V1.4-3D" | "V2.1" | "V3.0" /* version of SNNS */

SNNS                      "SNNS network definition file"  /* output file header */


/* eleven different headers */

GENERATED_AT              "generated at"
NETWORK_NAME              "network name :"
SOURCE_FILES              "source files"
NO.OF_UNITES              "no. of unites :"
NO.OF_CONNECTIONS         "no. of connections :"
NO.OF_UNIT_TYPES          "no. of unit types :"
NO.OF_SITE_TYPES          "no. of site types :"
LEARNING_FUNCTION         "learning function :"
PRUNING_FUNCTION          "pruning function :"
FF_LEARNING_FUNCTION      "subordinate learning function :"
UPDATE_FUNCTION           "update function :"


/* titles of the different sections */

UNIT_SECTION_TITLE        "unit definition section"
DEFAULT_SECTION_TITLE     "unit default section"
SITE_SECTION_TITLE        "site definition section"
TYPE_SECTION_TITLE        "type definition section"
CONNECTION_SECTION_TITLE  "connection definition section"
LAYER_SECTION_TITLE       "layer definition section"
SUBNET_SECTION_TITLE      "subnet definition section"
TRANSLATION_SECTION_TITLE "3D translation section"
TIME_DELAY_SECTION_TITLE  "time delay section"
```

```
/* column-titles of the different tables */

NO                       "no."
TYPE_NAME                "type name"
UNIT_NAME                "unit name"
ACT                      "act"
BIAS                     "bias"
ST                       "st"
POSITION                 "position"
SUBNET                   "subnet"
LAYER                    "layer"
ACT_FUNC                 "act func"
OUT_FUNC                 "out func"
SITES                    "sites"
SITE_NAME                "site name"
SITE_FUNCTION            "site function"
NAME                     "name"
TARGET                   "target"
SITE                     "site"
SOURCE:WEIGHT            "source:weight"
UNIT_NO                  "unitNo."
DELTA_X                  "delta x"
DELTA_Y                  "delta y"
Z                        "z"
LLN                      "LLN"
LUN                      "LUN"
TROFF                    "Troff"
SOFF                     "Soff"
CTYPE                    "Ctype"

INTEGER                  ["0"-"9"]+                                    /*integer */
SFLOAT                   ["+" | " " | "-"] ["1" | "0"] "." ["0"-"9"]{5} /*signed float*/
STRING                   ("A"-"Z" | "a"-"z" | "|")+                    /*string */
```

A.3.3 Grammar:

```
out_file               ::= file_header sections

file_header            ::= WHITESPACE  COMMENT h_snns EOL COMMENT h_generated_at EOL
                           COMMENT h_network_name EOL COMMENT h_source_files EOL
                           COMMENT h_no.of_unites EOL COMMENT h_no.of_connections EOL
                           COMMENT h_no.of_unit_types EOL COMMENT h_no.of_site_types EOL
                           COMMENT h_learning_function EOL COMMENT h_update_function EOL
                           COMMENT h_pruning_function EOL
                           COMMENT ff_learning_function EOL


/* parts of the file-header */

h_snns                 ::= SNNS BLANKS_TABS VERSION
h_generated_at         ::= GENERATED_AT BLANKS_TABS CSTRING
h_network_name         ::= NETWORK_NAME BLANKS_TABS STRING
h_source_files         ::= SOURCE_FILES [BLANKS_TABS COLON BLANKS_TABS CSTRING]
```

```
h_no.of_unites         ::= NO.OF_UNITES BLANKS_TABS INTEGER
h_no.of_connections    ::= NO.OF_CONNECTIONS BLANKS_TABS INTEGER
h_no.of_unit_types     ::= NO.OF_UNIT_TYPES BLANKS_TABS INTEGER
h_no.of_site_types     ::= NO.OF_SITE_TYPES BLANKS_TABS INTEGER
h_learning_function    ::= LEARNING_FUNCTION BLANKS_TABS STRING
h_pruning_function     ::= PRUNING_FUNCTION BLANKS_TABS STRING
h_ff_learning_function ::= FF_LEARNING_FUNCTION BLANKS_TABS STRING
h_update_function      ::= UPDATE_FUNCTION BLANKS_TABS STRING

sections               ::= COMMENT unit_section [COMMENT default_section]
                           [COMMENT site_section] [COMMENT type_section]
                           [COMMENT subnet_section] [COMMENT conn_section]
                           [COMMENT layer_section] [COMMENT trans_section]
                           [COMMENT time_delay_section] COMMENT


/* unit default section */

default_section        ::= DEFAULT_SECTION_TITLE CUT COMMENT WHITESPACE  default_block
default_block          ::= default_header SEVEN_COLUMN_LINE EOL
                           {COMMENT default_def} SEVEN_COLUMN_LINE EOL
default_header         ::= ACT COL_SEP BIAS COL_SEP ST COL_SEP SUBNET COL_SEP
                           LAYER COL_SEP ACT_FUNC COL_SEP OUT_FUNC CUT
default_def            ::= SFLOAT W_COL_SEP SFLOAT W_COL_SEP STRING W_COL_SEP
                           INTEGER W_COL_SEP INTEGER W_COL_SEP STRING W_COL_SEP
                           STRING CUT


/* site definition section */

site_section           ::= SITE_SECTION_TITLE CUT COMMENT WHITESPACE  site_block
site_block             ::= site_header TWO_COLUMN_LINE EOL {COMMENT site_def}
                           TWO_COLUMN_LINE EOL
site_header            ::= SITE_NAME SITE_FUNCTION CUT
site_def               ::= STRING W_COL_SEP STRING CUT


/* type definition section */

type_section           ::= TYPE_SECTION_TITLE CUT COMMENT WHITESPACE  type_block
type_block             ::= type_header FOUR_COLUMN_LINE EOL {COMMENT type_def}
                           FOUR_COLUMN_LINE EOL
type_header            ::= NAME COL_SEP ACT_FUNC COL_SEP OUT_FUNC COL_SEP SITES CUT
type_def               ::= STRING W_COL_SEP STRING W_COL_SEP STRING W_COL_SEP
                           [{STRING COMMA} STRING] CUT


/* subnet definition section */

subnet_section         ::= SUBNET_SECTION_TITLE CUT COMMENT WHITESPACE  subnet_block
subnet_block           ::= subnet_header TWO_COLUMN_LINE EOL {COMMENT subnet_def}
                           TWO_COLUMN_LINE EOL
subnet_header          ::= SUBNET COL_SEP UNIT_NO CUT
subnet_def             ::= INTEGER W_COL_SEP {INTEGER COMMA} INTEGER CUT
```

```
/* unit definition section /*

unit_section        ::= UNIT_SECTION_TITLE CUT COMMENT WHITESPACE  unit_block
unit_block          ::= unit_header TEN_COLUMN_LINE EOL {COMMENT unit_def}
                        TEN_COLUMN_LINE EOL

unit_header         ::= NO COL_SEP TYPE_NAME COL_SEP UNIT_NAME COL_SEP
                        ACT COL_SEP BIAS COL_SEP ST COL_SEP POSITION COL_SEP
                        ACT_FUNC COL_SEP OUT_FUNC COL_SEP SITES CUT
unit_def            ::= INTEGER W_COL_SEP ((STRING W_COL_SEP) | COL_SEP)
                        ((STRING W_COL_SEP) | COL_SEP) ((SFLOAT W_COL_SEP) | COL_SEP)
                        ((SFLOAT W_COL_SEP) | COL_SEP) ((STRING W_COL_SEP) | COL_SEP)
                        INTEGER COMMENT INTEGER COMMENT INTEGER W_COL_SEP
                        ((STRING W_COL_SEP) | COL_SEP) ((STRING W_COL_SEP) | COL_SEP)
                        [{STRING COMMA} STRING]

/* connection definition section */

connection_section  ::= CONNECTION_SECTION_TITLE CUT
                        COMMENT WHITESPACE connection_block
connection_block    ::= connection_header THREE_COLUMN_LINE EOL
                        {COMMENT connection_def} THREE_COLUMN_LINE EOL
connection_header   ::= TARGET COL_SEP SITE COL_SEP SOURCE:WEIGHT CUT
connection_def      ::= ((INTEGER W_COL_SEP) | COL_SEP) STRING W_COL_SEP
                        {INTEGER WHITESPACE  COLON WHITESPACE  SFLOAT COMMA}
                        INTEGER WHITESPACE  COLON WHITESPACE  SFLOAT CUT


/* layer definition section */

layer_section       ::= LAYER_SECTION_TITLE CUT COMMENT WHITESPACE  layer_block
layer_block         ::= layer_header TWO_COLUMN_LINE EOL {COMMENT layer_def}
                        TWO_COLUMN_LINE EOL
layer_header        ::= LAYER COL_SEP UNIT_NO CUT
layer_def           ::= INTEGER W_COL_SEP {INTEGER COMMENT} INTEGER CUT


/* 3D translation section */

translation_section ::= TRANSLATION_SECTION_TITLE CUT
                        COMMENT WHITESPACE  translation_block
translation_block   ::= translation_header THREE_COLUMN_LINE EOL
                        {COMMENT translation_def} THREE_COLUMN_LINE EOL
translation_header  ::= DELTA_X COL_SEP DELTA_Y COL_SEP Z CUT
translation_def     ::= INTEGER W_COL_SEP INTEGER W_COL_SEP INTEGER


/* time delay section */

td_section          ::= TIME_DELAY_SECTION_TITLE CUT COMMENT WHITESPACE  td_block
td_block            ::= td_header SIX_COLUMN_LINE EOL {COMMENT td_def}
                        SIX_COLUMN_LINE EOL
td_header           ::= NO COL_SEP LLN COL_SEP LUN COL_SEP
                        TROFF COL_SEP SOFF COL_SEP CTYPE CUT
td_def              ::= INTEGER W_COL_SEP INTEGER W_COL_SEP INTEGER W_COL_SEP
                        INTEGER W_COL_SEP INTEGER W_COL_SEP INTEGER W_COL_SEP
```

A.4 Grammar of the Pattern Files

The typographic conventions used for the pattern file grammar are the same as for the network file grammar (see section A.3.1.1).

A.4.1 Terminal Symbols

```
WHITE               {" " | "\t"}
FREE                {^"\n"}                   /* anything up to EOL */
COMMENT             "#" FREE "\n"
L_BRACKET           "["
R_BRACKET           "]"
V_NUMBER            [Vv]{INT}+"."{INT}+        /* version number */
INT                 ["0"-"9"]
NUMBER              [-+]?{{INT}+ | {INT}+{EXP} |{INT}+"."{INT}*({EXP})? |
                    {INT}*"."{INT}+({EXP})?}
EXP                 [Ee][-+]?{INT}+
VERSION_HEADER      "SNNS pattern definition file"
GENERATED_AT        "generated at" {FREE}* "\n"
NO_OF_PATTERN       "No. of patterns" {WHITE}* ":"
NO_OF_INPUT         "No. of input units" {WHITE}* ":"
NO_OF_OUTPUT        "No. of output units" {WHITE}* ":"
NO_OF_VAR_IDIM      "No. of variable input dimensions" {WHITE}* ":"
NO_OF_VAR_ODIM      "No. of variable output dimensions" {WHITE}* ":"
MAXIMUM_IDIM        "Maximum input dimensions" {WHITE}* ":"
MAXIMUM_ODIM        "Maximum output dimensions" {WHITE}* ":"
```

A.4.2 Grammar

```
pattern_file        ::= header pattern_list
header              ::= VERSION_HEADER V_NUMBER GENERATED_AT NO_OF_PATTERN NUMBER
                        i_head [o_head] [vi_head] [vo_head]
i_head              ::= [Comment] NO_OF_INPUT NUMBER
o_head              ::= [Comment] NO_OF_OUTPUT NUMBER
vi_head             ::= [Comment] NO_OF_VAR_IDIM NUMBER MAXIMUM_IDIM actual_dim
vo_head             ::= [Comment] NO_OF_VAR_ODIM NUMBER MAXIMUM_ODIM actual_dim
actual_dim          ::= (L_BRACKET actual_dim_rest R_BRACKET) |
                        (L_BRACKET R_BRACKET)
actual_dim_rest     ::= dim_entry | (actual_dim_rest dim_entry)
dim_entry           ::= INT
pattern_list        ::= pattern | (pattern_list pattern)
pattern             ::= pattern_start pattern_body
pattern_start       ::= [Comment] [actual_dim]
pattern_body        ::= pattern_entry | (pattern_body pattern_entry)
pattern_entry       ::= NUMBER
```

Appendix B

Example Network Files

The lines in the connection definition section have been truncated to 80 characters per line for printing purposes.

B.1 Example 1:

```
SNNS network definition file V3.0
generated at Fri Aug  3 00:28:44 1992

network name : klass
source files :
no. of units : 71
no. of connections : 610
no. of unit types : 0
no. of site types : 0


learning function : Std_Backpropagation
update function   : Topological_Order


unit default section :

act      | bias     | st | subnet | layer | act func     | out func
---------|----------|----|--------|-------|--------------|-------------
 0.00000 |  0.00000 | h  |      0 |     1 | Act_Logistic | Out_Identity
---------|----------|----|--------|-------|--------------|-------------


unit definition section :

no. | typeName | unitName | act      | bias     | st | position | act func | out func | sites
----|----------|----------|----------|----------|----|----------|----------|----------|-------
  1 |          | u11      |  1.00000 |  0.00000 | i  |  1, 1, 0 |||
  2 |          | u12      |  0.00000 |  0.00000 | i  |  2, 1, 0 |||
  3 |          | u13      |  0.00000 |  0.00000 | i  |  3, 1, 0 |||
  4 |          | u14      |  0.00000 |  0.00000 | i  |  4, 1, 0 |||
  5 |          | u15      |  1.00000 |  0.00000 | i  |  5, 1, 0 |||
  6 |          | u21      |  1.00000 |  0.00000 | i  |  1, 2, 0 |||
  7 |          | u22      |  1.00000 |  0.00000 | i  |  2, 2, 0 |||
  8 |          | u23      |  0.00000 |  0.00000 | i  |  3, 2, 0 |||
  9 |          | u24      |  1.00000 |  0.00000 | i  |  4, 2, 0 |||
 10 |          | u25      |  1.00000 |  0.00000 | i  |  5, 2, 0 |||
```

11		u31	1.00000	0.00000	i	1, 3, 0			
12		u32	0.00000	0.00000	i	2, 3, 0			
13		u33	1.00000	0.00000	i	3, 3, 0			
14		u34	0.00000	0.00000	i	4, 3, 0			
15		u35	1.00000	0.00000	i	5, 3, 0			
16		u41	1.00000	0.00000	i	1, 4, 0			
17		u42	0.00000	0.00000	i	2, 4, 0			
18		u43	0.00000	0.00000	i	3, 4, 0			
19		u44	0.00000	0.00000	i	4, 4, 0			
20		u45	1.00000	0.00000	i	5, 4, 0			
21		u51	1.00000	0.00000	i	1, 5, 0			
22		u52	0.00000	0.00000	i	2, 5, 0			
23		u53	0.00000	0.00000	i	3, 5, 0			
24		u54	0.00000	0.00000	i	4, 5, 0			
25		u55	1.00000	0.00000	i	5, 5, 0			
26		u61	1.00000	0.00000	i	1, 6, 0			
27		u62	0.00000	0.00000	i	2, 6, 0			
28		u63	0.00000	0.00000	i	3, 6, 0			
29		u64	0.00000	0.00000	i	4, 6, 0			
30		u65	1.00000	0.00000	i	5, 6, 0			
31		u71	1.00000	0.00000	i	1, 7, 0			
32		u72	0.00000	0.00000	i	2, 7, 0			
33		u73	0.00000	0.00000	i	3, 7, 0			
34		u74	0.00000	0.00000	i	4, 7, 0			
35		u75	1.00000	0.00000	i	5, 7, 0			
36		h1	0.99999	0.77763	h	8, 0, 0			
37		h2	0.19389	2.17683	h	8, 1, 0			
38		h3	1.00000	0.63820	h	8, 2, 0			
39		h4	0.99997	-1.39519	h	8, 3, 0			
40		h5	0.00076	0.88637	h	8, 4, 0			
41		h6	1.00000	-0.23139	h	8, 5, 0			
42		h7	0.94903	0.18078	h	8, 6, 0			
43		h8	0.00000	1.37368	h	8, 7, 0			
44		h9	0.99991	0.82651	h	8, 8, 0			
45		h10	0.00000	1.76282	h	8, 9, 0			
46		A	0.00972	-1.66540	o	11, 1, 0			
47		B	0.00072	-0.29800	o	12, 1, 0			
48		C	0.00007	-2.24918	o	13, 1, 0			
49		D	0.02159	-5.85148	o	14, 1, 0			
50		E	0.00225	-2.33176	o	11, 2, 0			
51		F	0.00052	-1.34881	o	12, 2, 0			
52		G	0.00082	-1.92413	o	13, 2, 0			
53		H	0.00766	-1.82425	o	14, 2, 0			
54		I	0.00038	-1.83376	o	11, 3, 0			
55		J	0.00001	-0.87552	o	12, 3, 0			
56		K	0.01608	-2.20737	o	13, 3, 0			
57		L	0.01430	-1.28561	o	14, 3, 0			
58		M	0.92158	-1.86763	o	11, 4, 0			
59		N	0.05265	-3.52717	o	12, 4, 0			
60		O	0.00024	-1.82485	o	13, 4, 0			
61		P	0.00031	-0.20401	o	14, 4, 0			
62		Q	0.00025	-1.78383	o	11, 5, 0			
63		R	0.00000	-1.61928	o	12, 5, 0			
64		S	0.00000	-1.59970	o	13, 5, 0			
65		T	0.00006	-1.67939	o	14, 5, 0			
66		U	0.01808	-1.66126	o	11, 6, 0			
67		V	0.00025	-1.53883	o	12, 6, 0			
68		W	0.01146	-2.78012	o	13, 6, 0			
69		X	0.00082	-2.21905	o	14, 6, 0			
70		Y	0.00007	-2.31156	o	11, 7, 0			
71		Z	0.00002	-2.88812	o	12, 7, 0			

```
connection definition section :

target | site | source:weight
-------|------|----------------------------------------------------------------
    36 |      |  1: 0.95093,  6: 3.83328, 11: 1.54422, 16: 4.18840, 21: 4.59526
                12: 2.30336, 17:-3.28721, 22:-0.43977, 27: 1.19506, 32:-0.84080
                23:-4.97246, 28:-3.30117, 33: 3.26851,  4:-0.19479,  9: 1.33412
                34:-1.02822,  5:-2.79300, 10:-1.97733, 15:-0.45209, 20:-0.61497
    37 |      |  1:-0.93678,  6: 0.68963, 11:-0.94478, 16:-1.06968, 21:-0.47616
                12: 2.62854, 17: 5.05391, 22:-0.37275, 27: 0.12598, 32: 0.27619
                23:-1.45917, 28:-1.97934, 33: 1.01118,  4: 4.39595,  9:-2.78858
                34:-0.14939,  5: 1.80792, 10: 3.66679, 15: 2.53150, 20:-1.07000
    38 |      |  1: 2.44151,  6: 0.41693, 11: 2.12043, 16: 1.40761, 21: 1.83566
                12:-0.55002, 17: 2.08524, 22: 0.63304, 27: 0.27301, 32:-2.49952
                23: 3.14177, 28:-1.25889, 33:-6.35069,  4:-5.25082,  9: 0.01774
                34:-3.66092,  5: 3.24401, 10: 1.88082, 15: 6.44985, 20: 3.24165
    39 |      |  1: 5.17748,  6:-4.45709, 11:-0.65733, 16:-2.26190, 21:-2.69957
                12:-1.43420, 17: 0.33409, 22:-0.74423, 27:-1.38010, 32: 3.08174
                23:-4.42961, 28:-1.09858, 33: 2.09879,  4:-1.30835,  9: 0.79940
                34: 1.99276,  5: 2.61433, 10:-3.56919, 15: 1.00952, 20: 2.86899
    40 |      |  1: 3.03612,  6: 0.05247, 11:-3.20839, 16:-4.03382, 21:-3.55648
                12: 0.23398, 17: 1.33895, 22: 6.03206, 27:-0.01723, 32: 0.09160
                23:-1.07894, 28:-1.77930, 33: 1.59529,  4:-1.57236,  9: 0.74423
                34:-0.13875,  5: 5.30719, 10: 2.13168, 15:-2.34832, 20:-5.00616
    41 |      |  1:-4.41380,  6:-1.48152, 11:-2.62748, 16:-1.00557, 21:-0.06430
                12: 3.93844, 17:-4.01591, 22: 0.76102, 27:-0.36823, 32: 0.54661
                23: 4.15954, 28: 2.96118, 33:-3.30219,  4:-0.24202,  9: 1.56077
                34:-0.20287,  5:-1.46062, 10: 1.79490, 15: 1.96920, 20: 3.72459
    42 |      |  1: 1.97383,  6: 2.53253, 11: 2.04922, 16: 1.13969, 21: 1.81064
                12: 0.32565, 17: 4.64358, 22: 1.02883, 27:-1.05720, 32:-0.71916
                23:-1.00499, 28:-1.10925, 33:-3.18685,  4: 2.12575,  9: 0.36763
                34:-0.18372,  5:-4.93490, 10: 0.26375, 15:-2.02860, 20:-5.43881
    43 |      |  1: 0.07183,  6:-2.69081, 11:-1.24533, 16:-2.01347, 21:-1.36689
                12:-2.11356, 17: 1.24788, 22: 1.23107, 27: 0.27674, 32:-2.45891
                23: 5.17387, 28: 1.68170, 33:-2.30420,  4: 2.17011,  9: 0.86340
                34:-2.23131,  5:-0.11916, 10:-4.39609, 15:-2.92706, 20:-5.43783
    44 |      |  1:-1.27907,  6: 1.89325, 11:-0.60419, 16: 3.60368, 21: 4.24280
                12:-2.77766, 17: 1.01698, 22:-1.97236, 27: 1.38773, 32:-2.55429
                23: 1.95344, 28: 2.85157, 33:-0.55796,  4:-0.64082,  9: 1.92937
                34:-2.71524,  5: 5.31087, 10:-2.08897, 15:-5.75332, 20: 2.43438
    45 |      |  1:-1.22455,  6: 0.92594, 11: 1.13199, 16:-1.65062, 21:-1.41481
                12:-3.04575, 17:-3.21280, 22:-0.23726, 27: 2.11836, 32: 2.23237
                23: 5.96261, 28: 2.00822, 33: 2.97409,  4: 3.90943,  9: 1.54990
                34: 2.42877,  5:-3.58017, 10: 2.31309, 15:-4.01833, 20: 0.28834
    46 |      | 45:-3.97560, 44: 0.45729, 43:-1.16526, 42: 0.38939, 41: 2.80876
                36:-4.04184
    47 |      | 45:-4.88750, 44:-3.33955, 43:-1.72110, 42: 0.94756, 41:-2.24993
                36: 0.14327
    48 |      | 45: 1.02597, 44:-1.82773, 43:-1.04974, 42: 2.09881, 41:-0.53220
                36: 2.75042
    49 |      | 45: 1.58579, 44:-3.38572, 43:-0.89166, 42: 2.86233, 41: 2.25429
                36: 1.92163
    50 |      | 45:-2.95134, 44: 2.39376, 43:-2.95486, 42:-0.11771, 41:-2.41775
                36:-0.73749
    51 |      | 45:-4.16732, 44: 2.19092, 43: 3.46879, 42: 0.44175, 41:-2.47295
                36:-0.40437
    52 |      | 45: 1.78256, 44: 4.64443, 43:-2.50408, 42: 0.65889, 41:-2.52796
                36:-1.73887
    53 |      | 45:-3.64449, 44: 2.60025, 43:-1.57915, 42:-0.18638, 41:-4.14214
                36:-4.29717
    54 |      | 45:-0.45205, 44:-1.44890, 43: 5.23345, 42:-0.35289, 41: 2.43160
                36:-1.99719
    55 |      | 45: 0.46855, 44:-2.84431, 43:-1.80938, 42:-4.49606, 41: 1.16736
                36:-4.07946
```

```
   56 |      | 45:-1.27961, 44: 0.81393, 43: 2.66027, 42:-1.05007, 41: 0.47655
                 36: 0.72057
   57 |      | 45:-1.21679, 44:-3.13145, 43:-0.69538, 42: 0.05268, 41:-3.12564
                 36: 2.16523
   58 |      | 45:-3.44107, 44: 2.18362, 43:-1.60547, 42:-0.50213, 41: 2.47751
                 36: 1.05544
   59 |      | 45:-5.06022, 44:-5.08796, 43:-0.74552, 42:-0.67009, 41: 2.90942
                 36: 1.10170
   60 |      | 45:-0.03838, 44:-2.59148, 43:-0.98185, 42:-4.97460, 41: 1.03714
                 36: 4.49324
   61 |      | 45:-3.65178, 44:-2.61389, 43:-1.76429, 42: 1.01853, 41:-3.90286
                 36: 0.46651
   62 |      | 45: 1.17767, 44:-3.70962, 43:-1.11298, 42:-2.77810, 41: 2.22540
                 36:-0.65937
   63 |      | 45: 1.52270, 44:-2.42352, 43: 2.59287, 42: 0.19606, 41:-3.00602
                 36:-3.14334
   64 |      | 45: 3.00298, 44:-3.65709, 43:-1.65150, 42: 1.44547, 41:-3.98775
                 36:-4.02590
   65 |      | 45: 0.56820, 44: 2.37122, 43: 2.89143, 42:-4.22975, 41:-2.32045
                 36:-2.17370
   66 |      | 45:-4.13007, 44:-0.30654, 43:-0.63380, 42:-5.63405, 41:-1.78425
                 36: 1.46460
   67 |      | 45: 1.66534, 44: 1.99220, 43:-1.22676, 42:-4.09076, 41:-3.58451
                 36: 1.40745
   68 |      | 45: 0.60032, 44: 2.75100, 43:-1.17663, 42:-4.25699, 41: 1.60600
                 36:-4.23531
   69 |      | 45:-4.31415, 44:-3.41717, 43:-0.77030, 42:-1.36290, 41: 1.92319
                 36: 0.28689
   70 |      | 45: 0.54085, 44: 1.80036, 43:-2.19014, 42:-1.20720, 41: 1.54519
                 36:-2.79769
   71 |      | 45: 0.00018, 44:-4.11360, 43: 0.85345, 42: 1.19947, 41: 1.22067
                 36:-3.29634
-------|------|----------------------------------------------------------------
```

B.2 Example 2:

```
SNNS network definition file V3.0
generated at Fri Aug  3 00:25:42 1992

network name : xor
source files :
no. of units : 4
no. of connections : 5
no. of unit types : 2
no. of site types : 2


learning function : Quickprop
update function   : Topological_Order


site definition section :

 site name | site function
-----------|------------------
 inhibit   | Site_Pi
 excite    | Site_WeightedSum
-----------|------------------
```

```
type definition section :

name          | act func     | out func     |sites
--------------|--------------|--------------|---------
outType       | Act_Logistic | Out_Identity |
LongeroutType | Act_Logistic | Out_Identity |
--------------|--------------|--------------|---------


unit default section :

act      | bias     | st | subnet | layer | act func     | out func
---------|----------|----|--------|-------|--------------|-------------
 0.00000 |  0.00000 | h  |      0 |     1 | Act_Logistic | Out_Identity
---------|----------|----|--------|-------|--------------|-------------


unit definition section :

no. | typeName      | unitName | act      | bias     | st | position | act func
----|---------------|----------|----------|----------|----|----------|---------
  1 |               | in_1     |  1.00000 |  0.00000 | i  | 3,5,0    |||
  2 |               | in_2     |  1.00000 |  0.00000 | i  | 9,5,0    |||
  3 |               | hidden   |  0.04728 | -3.08885 | h  | 6,3,0    |||
  4 |               | result   |  0.10377 | -2.54932 | o  | 6,0,0    |||
----|---------------|----------|----------|----------|----|----------|---------


connection definition section :

target | site    | source:weight
-------|---------|------------------------------------------------------------
     3 |         | 1: 4.92521, 2:-4.83963
     4 |         | 1:-4.67122, 2: 4.53903, 3:11.11523
-------|---------|------------------------------------------------------------
```

Appendix C

Example Snnsbat Protocol File

```
SNNS Kernel V3.3 Batchlearning Program
Configuration file: 'newbat.config'
Log file          : 'test.log2'

Networkfile '/usr/local/bv/SNNS/SNNSv3.3/examples/letters.net' loaded.
Network name: klass
No. of units: 71
No. of sites: 0
No. of links: 610
Learning Function: Quickprop
2 Learning Parameters

Learning Parameter #1: 0.200000
Learning Parameter #2: 0.300000

Patternfile '/usr/local/bv/SNNS/SNNSv3.3/examples/letters.pat' loaded.
No. of patterns: 26

No. of cycles: 5

Max. network error to stop: 0.100000

Patterns are shuffled


Test pattern file : '/usr/local/bv/SNNS/SNNSv2.1/examples/letters.pat'
Initialization Function: Randomize_Weights
2 Initialization Parameters

Initialization Parameter #1: -1.000000
Initialization Parameter #2: 1.000000


Result File              : 'letters.res'
Result File Start Pattern: 1
Result File End Pattern  : 26
Result File Output Pattern included


*******************************************************

SNNS Kernel V3.3 Batchlearning started at Tue Jul 28 17:04:55 1992


Network initialized with
Randomize_Weights -1.00 1.00
```

```
Cycle: 0
Learning function value(s): [1]: 96.6988

Cycle: 1
Learning function value(s): [1]: 19.7769

Cycle: 2
Learning function value(s): [1]: 12.6723

Cycle: 3
Learning function value(s): [1]: 12.6058

Cycle: 4
Learning function value(s): [1]: 12.5618
Test Pattern File '/usr/local/bv/SNNS/SNNSv2.1/examples/letters.pat' loaded.
No. of test patterns: 26


Result file saved.

Network saved to trained_letters.net.

SNNS Kernel V3.3 Batchlearning terminated at Tue Jul 28 17:04:56 1992
System: SunOS  Node: monet  Machine: sun4c
********************************************************

----   STATISTICS  ----
No. of learned cycles: 5
No. of units updated : 9230
No. of sites updated : 0
No. of links updated : 79300
CPU Time used: 1.95 seconds
User time: 1 seconds
No. of connection updates per second (CUPS): 4.066667e+04
```

Bibliography

[Aka69] H. Akaike. *Annals of the institute of statistical mathematics, 21*, chapter Fitting autoregressive models for prediction. 1969.

[Ama89] Shun-Ichi Amari. *Characteristics of Encoded Associative Memory.* Springer Verlag 1989, 1989.

[BD95] Michael R. Berthold and Jay Diamond. Boosting the performance of rbf networks with dynamic decay adjustment. In G. Tesauro, D. S. Touretzky, and T.K. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, 1995.

[BH93] D. Stork B. Hassibi. Second order derivatives for network pruning: Optimal Brain Surgeon. In T. J. Sejnowski G. E. Hinton and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems (NIPS) 5*, pages 164–171, San Mateo, 1993. Morgan Kaufmann Publishers Inc.

[Bie94] J. Biedermann. Anwendungen Neuronaler Netze beim VLSI-CAD. Diplomarbeit, Institut für Numerische und Angewandte Mathematik, Georg-August-Universität Göttingen, 1994.

[CG87a] G. A. Carpenter and S. Grossberg. A Massively Parallel Architecture for a Selforganizing Neural Pattern Recognition Machine. *Computer Vision, Graphics and Image Processing*, 37:54–115, 1987.

[CG87b] G. A. Carpenter and S. Grossberg. Stable self-organization of pattern recognition codes for analog input patterns. *Applied Optics*, 26:4919–4930, 1987.

[CG91] G. A. Carpenter and S. Grossberg. ARTMAP: Supervised Real-Time Learning and Classification of Nonstationary Data by a Self-Organizing Neural Network. *Neural Networks*, 4:543–564, 1991.

[D.E93] D.Elliott. A better activation function for artificial neural networks. ISR Technical Report TR 93-8, University of Maryland, 1993.

[DH73] R. Duda and P. Hart. *Pattern Classification and Scene Analysis.* Wiley & Sons, Inc, 1973.

[Elm90] J. L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[Fah88] Scott E. Fahlman. Faster-learning variations on back-propagation: An empirical study. In T. J. Sejnowski G. E. Hinton and D. S. Touretzky, editors, *1988 Connectionist Models Summer School*, San Mateo, CA, 1988. Morgan Kaufmann.

[Fah91] S. E. Fahlman. The recurrent cascade-correlation architecture. Technical Report CMU-CS-91-100, School of Computer Science, Carnegie Mellon University, 1991.

[FL91] S. E. Fahlman and C. Lebiere. The cascade-correlation learning architecture. Technical Report CMU-CS-90-100, School of Computer Science, Carnegie Mellon University, August 1991.

[GFL86] N.H. Goddard, M.A. Fanty, and K. Lynne. The Rochester Connectionist Simulator. Technical Report 189, Computer Science Department, Univ. of Rochester, June 1986.

[GHW79] G. H. Golub, M. Heath, and G. Wahba. *Technometrics, 21*, chapter Generalized cross-calidation as a method for choosing ridge parameter. 1979.

[GLML89] N. Goddard, K.J. Lynne, T. Mintz, and L.Bukys. The Rochester Connectionist Simulator. Technical Report 233 (revised), Univ. of Rochester, NY, oct 1989.

[God87] N. Goddard. *The Rochester Connectionist Simulator: User Manual.* Univ. of Rochester, NY, 1987.

[Har83] S. Harrington. *Computer Graphics - A Programming Approach.* McGraw-Hill, 1983.

[Her92] K.-U. Herrmann. ART – Adaptive Resonance Theory – Architekturen, Implementierung und Anwendung. Diplomarbeit 929, IPVR, Universität Stuttgart, 1992.

[HF91] M. Hoefeld and S. E. Fahlman. Learning with limited numerical precision using the cascade-correlation algorithm. Technical Report CMU-CS-91-130, School of Computer Science, Carnegie Mellon University, 1991.

[Hil85] W.D. Hillis. *The Connection Machine.* MIT Press, 1985.

[HS86a] W.D. Hillis and G.L. Steele. Data parallel algorithms. *ACM*, 29(12):1170–1183, 1986.

[HS86b] W.D. Hillis and G.L. Steele. Massively parallel computers: The Connection Machine and NONVON. *Science*, 231(4741):975–978, 1986.

[Hüb92] R. Hübner. 3d–Visualiserung der Topologie und der Aktivität neuronaler Netze. Diplomarbeit 846, IPVR, Universität Stuttgart, 1992.

[Hud92] M.J. Hudak. Rce classifiers: Theory and practice. *Cybernetics and Systems*, pages 483–515, 1992.

[JGHL80] G. G. Judge, W. E. Griffiths, R. C. Hill, and T. Lee. *The theory and practice of econometrics.* John Wiley & sons, 1980.

[Jor86a] M. I. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, pages 531–546, Hillsdale NJ, 1986. Erlbaum.

[Jor86b] M. I. Jordan. Serial order: A parallel distributed processing approach. Technical Report Nr. 8604, Institute for Cognitive Science, University of California, San Diego, La Jolla, California, 1986.

[JS91] R. Dow J. Sietsma. Creating Artificial Neural Networks That Generalize. *Neural Networks*, 4(1):67–79, 1991.

[KKLT92] T. Kohonen, J. Kangas, J. Laaksoonen, and K. Torkkola. Lvq_pak learning vector quantization program package. Technical report, Laboratory of Computer and Information Science Rakentajanaukio 2 C, 1991 - 1992.

[KL90] J. Kindermann and A. Linden. Inversion of neural networks by gradient descent. *Parallel Computing*, 14:277–286, 1990.

[Koh88] T Kohonen. *Self-Organization and Associative Memory.* Springer-Verlag, 1988.

[Koh89] Teuvo Kohonen. *Self-Organization and Associative Memory, Third Edition.* Springer Verlag 1989, 1989.

[Kor89] T. Korb. Entwurf und Implementierung einer deklarativen Sprache zur Beschreibung neuronaler Netze. Studienarbeit 789, IPVR, Universität Stuttgart, 1989.

[Kub91] G. Kubiak. Vorhersage von Börsenkursen mit neuronalen Netzen. Diplomarbeit 822, IPVR, Universität Stuttgart, 1991.

[KZ89] T. Korb and A. Zell. A declarative neural network description language. In *Microprocessing and Microprogramming.* North-Holland, August 1989.

[Mac90] N. Mache. Entwurf und Realisierung eines effizienten Simulatorkerns für neuronale Netze. Studienarbeit 895, IPVR, Universität Stuttgart, 1990.

[Mam92] G. Mamier. Graphische Visualisierungs–Hilfsmittel für einen Simulator neuronaler Netze. Diplomarbeit 880, IPVR, Universität Stuttgart, 1992.

[MM89] P. Smolensky M. Mozer. Skeletonization: A Technique for Trimming the Fat from a Network via Relevance Assessment. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems (NIPS) 1*, pages 107–115, San Mateo, 1989. Morgan Kaufmann Publishers Inc.

[MP69] M. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry.* The MIT Press, Cambridge, Massachusetts, 1969.

[MR92] H.Braun M. Riedmiller. Rprop: A fast adaptive learning algorithm. In *Proc. of the Int. Symposium on Computer and Information Science VII*, 1992.

[MR93] H.Braun M. Riedmiller. Rprop: A fast and robust backpropagation learning strategy. In *Proc. of the ACNN*, 1993.

[Pet91] A. Petzold. Vergleich verschiedener Lernverfahren für neuronale Netze. Studienarbeit 940, IPVR, Universität Stuttgart, 1991.

[RB93] M Riedmiller and H Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Proceedings of the IEEE International Conference on Neural Networks 1993 (ICNN 93)*, 1993.

[RCE82] D.L. Reilly, L.N. Cooper, and C. Elbaum. A neural model for category learning. *Biol. Cybernet.*, 45, 1982.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the microstructure of cognition; Vol. 1: Foundations*, Cambridge, Massachusetts, 1986. The MIT Press.

[Rie93] M Riedmiller. Untersuchungen zu konvergenz und generalisierungsverhalten überwachter lernverfahren mit dem SNNS. In *Proceedings of the SNNS 1993 workshop*, 1993.

[RM86] D.E. Rumelhart and J.L. McClelland. *Parallel Distributed Processing*, volume 1. MIT Press, 1986.

[Sch78] G. Schwarz. *Annals of statistics 6*, chapter Estimating the dimensions of a model. 1978.

[Sch91a] M. Schmalzl. Rotations- und translationsinvariante Erkennung von maschinengeschrieben Zeichen mit neuronalen Netzen. Studienarbeit 1011, IPVR, Universität Stuttgart, 1991.

[Sch91b] D. Schmidt. Anwendung neuronaler Netzwerkmodelle zur Erkennung und Klassifikation exogener und endogener Komponenten hirnelektrischer Potentiale. Studienarbeit 1010, IPVR, Universität Stuttgart, 1991.

[Sch94] T. Schreiner. Ausdünnungsverfahren für Neuronale Netze. Diplomarbeit 1140, IPVR, Universität Stuttgart, 1994.

[Shi68] R. Shibata. *Biometrika, 68*, chapter An optimal selection of Regression Variables. 1968.

[Sie91] J. Sienel. Kompensation von Störgeräuschen in Spracherkennungssystemen mittels neuronaler Netze. Studienarbeit 1037, IPVR, Universität Stuttgart, 1991.

[SK92] J. Schürmann and U. Kreßel. Mustererkennung mit statistischen Methoden. Technical report, Daimler-Benz AG, Forschungszentrum Ulm, Institut für Informatik, 1992.

[Som89] T. Sommer. Entwurf und Realisierung einer graphischen Benutzeroberfläche für einen Simulator konnektionistischer Netzwerke. Studienarbeit 746, IPVR, Universität Stuttgart, 1989.

[Soy93] T. Soyez. Prognose von Zeitreihen mit partiell recurrenten Netzen und Backpropagtion. Studienarbeit 1270, IPVR, University of Stuttgart, 1993.

[SUN86] SUN. Sunview user reference manual. Technical report, SUN microsystems, 1986.

[Vei91] A. Veigel. Rotations– und rotationsinvariante Erkennung handgeschriebener Zeichen mit neuronalen Netzwerken. Diplomarbeit 811, IPVR, Universität Stuttgart, 1991.

[Vog92] M. Vogt. Implementierung und Anwendung von 'Generalized Radial Basis Functions' in einem Simulator neuronaler Netze. Diplomarbeit 875, IPVR, Universität Stuttgart, 1992.

[Was89] Philip D. Wasserman. *Neural Computing.* Van Nostrand Reinhold, New York, 1989.

[Weh94] C. Wehrfritz. Neuronale netze als statistische methode zur erklärung von klassifikationen. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Statistik 1, 1994.

[Wer88] P. Werbos. Backpropagation: Past and future. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 343–353. IEEE Press, 1988.

[WHH+89] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. Lang. Phoneme Recognition Using Time Delay Neural Networks. *IEEE Transactions on Accoustics, Speech and Signal Processing*, 37:328–339, 1989.

[YLC90] S. Solla Y. Le Cun, J. Denker. Optimal Brain Damage. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems (NIPS) 2*, pages 598–605, San Mateo, 1990. Morgan Kaufmann Publishers Inc.

[You89] D.A. Young. *X-Window-System - Programming and Applications with Xt.* Prentice Hall, 1989.

[Zel94] Andreas Zell. *Simulation Neuronaler Netze.* Addison-Wesley, 1994. published in German.

[Zim91] P. Zimmerer. Translations– und rotationsinvariante Erkennung von Werkstücken mit neuronalen Netzwerken. Diplomarbeit 777, IPVR, Universität Stuttgart, 1991.

[Zip90] D. Zipser. Subgrouping reduces compexity and speeds up learning in recurrent networks. In D.S. Touretzky, editor, *Advances in Neural Information Processing systems II*, pages 638–641, San Mateo, California, 1990. Morgan Kaufmann.

[ZKSB89] A. Zell, T. Korb, T. Sommer, and R. Bayer. Netsim: Ein Simulator für neuronale Netze. In *GWAI-89.* Springer-Verlag (Informatik-Fachberichte), 1989.

[ZMS90] A. Zell, N. Mache, and T. Sommer. Applications of neural networks. In *Proc. Applications of Neural Networks Conf., SPIE*, volume 1469, pages 535–544, Orlando Florida, 1990. Aerospace Sensing Intl. Symposium.

[ZMSK91a] A. Zell, N. Mache, T. Sommer, and T. Korb. Design of the SNNS neural network simulator. In *Östreichische Artificial-Intelligence-Tagung*, pages 93–102, Wien, 1991. Informatik-Fachberichte 287, Springer Verlag.

[ZMSK91b] A. Zell, N. Mache, T. Sommer, and T. Korb. Recent Developments of the SNNS Neural Network Simulator. In *Proc. Applications of Neural Networks Conf., SPIE*, volume 1469, pages 708–719, Orlando Florida, 1991. Aerospace Sensing Intl. Symposium.

[ZMSK91c] A. Zell, N. Mache, T. Sommer, and T. Korb. The SNNS Neural Network Simulator. In *GWAI-91, 15. Fachtagung für künstliche Intelligenz*, pages 254–263. Informatik-Fachberichte 285, Springer Verlag, 1991.

[ZZ91] P. Zimmerer and A. Zell. Translations– und rotationsinvariante Erkennung von Werkstücken mit neuronalen Netzwerken. In *Informatik-Fachberichte 290*, pages 51–58, München, 1991. DAGM Symposium.

Index