University of Stuttgart

Institute for Parallel and Distributed High Performance Systems
Breitwiesenstr. 20–22
7000 Stuttgart 80
Fed. Rep. of Germany

Andreas Zell
Niels Mache, Ralf Hübner
Günter Mamier, Michael Vogt
Kai-Uwe Herrmann, Michael Schmalzl
Tilman Sommer, Artemis Hatzigeorgiou
Sven Döring, Dietmar Posselt

external contributions by
Martin Reczko, Martin Riedmiller

SNNS

Stuttgart Neural Network Simulator

User Manual, Version 3.0

Report No. 3/93

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 Universität 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 two main components that are depicted in figure 1.1: Simulator kernel and graphical user interface. 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.

During the development of the user interface, high consideration was given to its effective handling. Thus 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.

Another important point 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 with a layer technique.

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 the `twm` window manager.

This document is structured as follows:

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

[1] X Graphical User Interface

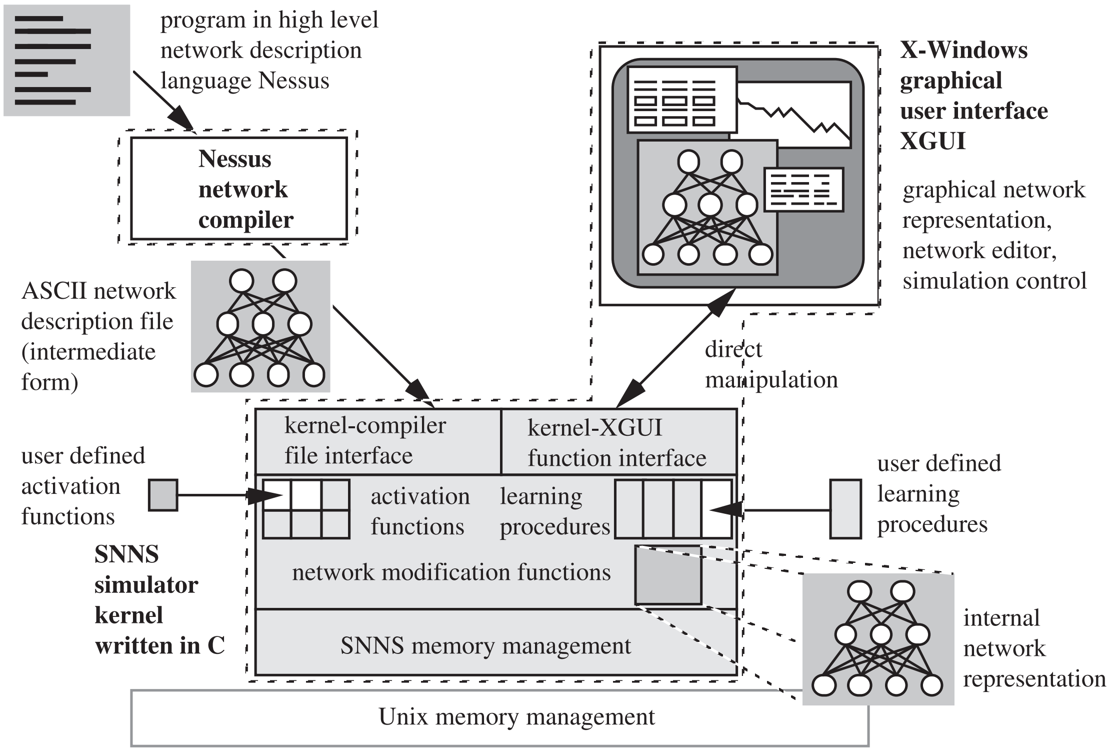

Figure 1.1: SNNS components: simulator kernel, graphical user interface, and network compiler

machine type	**operating system**
SUN SparcSt. SLC, ELC,IPC	SunOS 4.1.2
SUN SparcSt. 2 GX, GS	SunOS 4.1.2
SUN SparcSt. 10	SunOS 4.1.2
DECstation 2100, 3100	Ultrix V4.2
DECstation 5000	Ultrix V4.2
IBM-PC 80386, 80486	Interactive Unix
IBM-PC 80486	Linux
IBM RS 6000/320, 320H	AIX V3.1, AIX V3.2
IBM RS 6000/530H	AIX V3.1, AIX V3.2
HP 9000/720, 730	HP-UX 8.07

Table 1.1: Machines and operating systems on which SNNS has been tested (as of March 1993)

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

Chapter 6 is about a tool to facilitate the generation of large, regular networks from the graphical user interface.

Chapter 7 describes the network analyzing facility, built into SNNS. The analyzing method is called inversion.

Chapter 8 describes the connectionist models that are already implemented in SNNS, with a strong emphasis on the less familiar network models.

Chapter 9 introduces a new visualization component for three-dimensional visualization of the topology and the activity of neural networks with wireframe or solid models.

Chapter 10 introduces the batch capabilities of SNNS. They can be accessed via an additional interface to the kernel, that allows for easy background execution.

Chapter 11 describes the structure of the SNNS simulator kernel.

Chapter 12 discusses the internal data structures used for describing the nets.

Chapter 13 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 with it.

Chapter 14 details the activation functions and output function which are already built in.

Chapter 15 gives implementation details of the simulator kernel.

Chapter 16 deals with the concepts used in the implementation of SNNS-XGUI. It also explains the source code, which is especially interesting for further projects.

Chapter 17 gives implementation details of the 3D network visualization component.

The description of the Nessus programming language and compiler is not included in this document but in a separate manual. This manual is currently only available in German. It consists of the description of the language elements, which are explained with examples, and information regarding the implementation of the compiler.

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, the Nessus network compiler or even an editor.

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

Chapter 2

Licensing, Copying and Acknowledgements

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.

[1]After July 1st, the zip code will change to 70565 Stuttgart

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, sublicense, distribute or transfer SNNS except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, 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 ususally 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

```
SNNSv3.0.tar.Z
```

or in several parts as files

```
SNNSv3.0.tar.Z.aa, SNNSv3.0.tar.Z.ab, ...
```

These split files are each less than 1 MB and can be joined with the Unix 'cat' command into one file `SNNSv3.0.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 SNNSv3.0.tar.Z
```

and then use the command

```
tar -xvf SNNSv3.0.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.

SNNS currently runs on colour or black and white Sun-3, Sun-4 (SPARC) systems under SunOS 4.1.1 with X-Windows X11R4 and X11R5 (Athena widget set, twm, MIT fonts etc.) and under Sun OpenWindows 3.0. It also runs on DecStations with MIPS R3000 Processor with X-Windows X11R4 and X11R5. It has not been tested on out of the box DecStations with DecWindows. It has been tested on the IBM RS 6000 with AIX version 3.1 and on HP 9000/700 machines with HP-UX 8.07. In general, the SNNS kernel will run on almost any Unix system, while the graphical user interface might give problems with systems which are not fully X11R4 (or X11R5) compatible.

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. 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 subdirectoires 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 architecure, e.g. `sun3`, `sparc`, `dec`, `rs6000`, `hp` 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
```

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

2.4 Acknowledgements

SNNS is a joint effort of a number of people, most of them computer science students 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 a 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 3.3MB in 110.000 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:

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 SNNS simulator 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	Implementation of Radial Basis Functions and application: use of RBF's in SNNS [Vog92]. Together with Günter Mamier implementation of Time Delay Networks.
Günter Mamier	SNNS visualisation and analyzing tools [Mam92]. Implementation of the batch execution cabability. Compilation and continous update of the user manual.

Michael Schmalzl	SNNS network creation tool Bignet, implementation of Cascade Correlation, and printed character recognition with SNNS [Sch91a]
Kai-Uwe Herrmann	Implementation of the ART models ART1, ART2, ARTMAP and modification of the BigNet tool [Her92].
Artemis Hatzigeorgiou	Video documentation about the SNNS project, SNNS simulator learning procedure Backpercolation 1.[2]
Andreas Veigel	Application: handwritten character recognition with neural networks [Vei91]
Peter Zimmerer	Application: Position- and rotation-invariant recognition of flat machine parts with neural networks [Zim91], [ZZ91]
Dieter Schmidt	Application: Classification of endogenic and exogenic components of EEG signals with neural networks [Sch91b]
Jürgen Sienel	Application: Noise reduction in speech recognition systems with neural networks [Sie91]
Günther Kubiak	Application: Prediction of stock values with neural networks [Kub91]
Martin Riedmiller	Implementation of RPROP in SNNS
Martin Reczko	Implementation of Backpropagation Through Time (BPTT), Batch BPTT (BBPTT), and Quickprop Through Time (QPTT).
Sven Döring	ANSI-C translation of SNNS.
Dietmar Posselt	ANSI-C translation of SNNS.

Günter Mamier and Andreas Zell translated the German original of this documentation into English.

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] with the old SunView Interface. This heritage can still be detected in the user interface.

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] 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.

2.5 New Features of Release 3.0

Users already familiar with SNNS and its usage may be interested in the differences between the versions 2.1 and 3.0. New users of SNNS may skip this section and proceed with the next chapter.

1. Complete translation from K&R-C to ANSI-C. An ANSI-C compiler is now required to compile SNNS[3].
2. Implementation of ART1, ART2, and ARTMAP network models.
3. Implementation of Time Delay Networks (TDNNs).
4. Implementation of the RPROP learning algorithm.
5. Implementation of the learning algorithms Backpropagation Through Time (BPTT), Batch Backpropagation Through Time (BBPTT), and Quickprop Through Time (QPTT) for recurrent networks.
6. Implementation of cascade correlation and recurrent cascade correlation.
7. The bignet panel was split up. Now there is one panel for each special network class.
8. Change in the kernel-xgui interface: the function `krui_delete_Unit` was replaced by `krui_delete_UnitList`.
9. The kernel now handles 10 learning parameters instead of 5.
10. The tool snnsbat now handles multiple training / callback runs instead of just one.
11. This user manual was expanded and got an index.
12. In the 3D display the units are now shaded even when the activation is color coded.
13. The extra stop button in the remote panel was removed.
14. Bugs removed:
    - The wrong definition 'extern int fprintf()' in the file ui_funcdispl.h was removed.
    - The name of the learning function in the remote panel is now erased when a new network is loaded.
    - All patterns become invalid when the network structure is changed after patterns have been loaded. Old patterns caused segmentation faults so far.
    - The button `DEF` in the info panel was not working propperly. This is fixed now.
    - An open weight-display caused segmentation faults after changes to the network. This should be resolved now.
    - The file interface didn't like the description of sites in the 2.1 release. We convinced it to behave as expected.

[3]This was a major change of SNNS, especially in the graphical user interface. It uncovered a number of bugs. We hope, that it did not introduce too many new ones.

- When links were set to xx.yyyy in the info panel, they were reported as being xx.yyyy8. The 8 is removed now (the link actually was xx.yyyy).
- Network file consistency test was improved.
- Unit default types in the network file are now treated correctly.
- An allocation of 0 units is now possible. (Produced a segmentation fault under AIX up to now).
- Some errors in the user manual have been corrected.

The translation of SNNS from K&R C to ANSI-C uncovered several unknown bugs. All those have been removed now. We hope that the translation improves the overall robustness of SNNS and that not too many new bugs have been created.

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 modelling 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].

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.

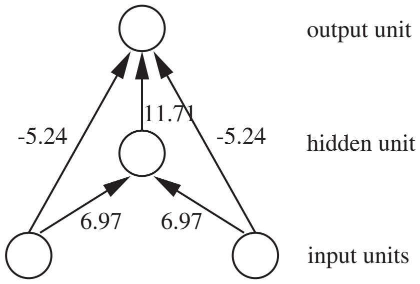

Figure 3.1: A small network with three layers of units

The actual information processing within the units is modelled 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 modelled explicitly (i.e. there is no propagation delay or explicit modelling 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. Different sites can have different site functions. This selective treatment of incomming information allows more powerful connectionist models. Picture 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 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).

[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.

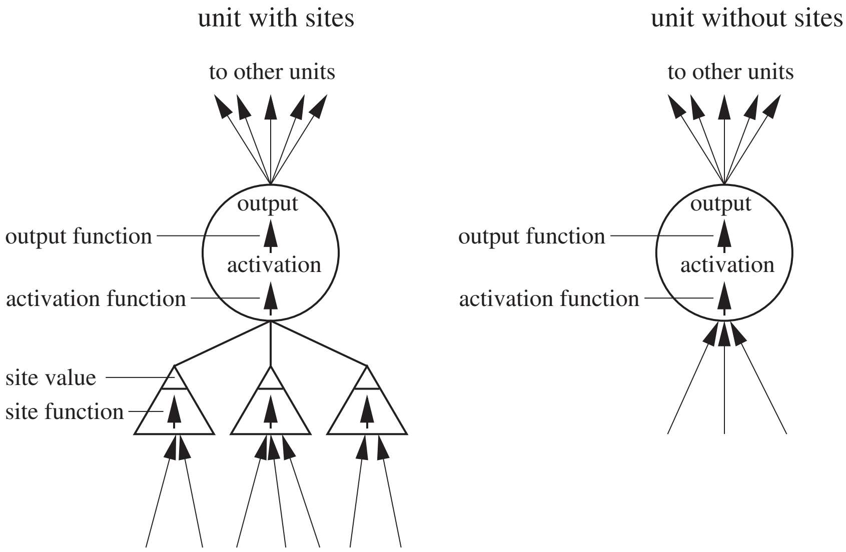

Figure 3.2: One unit with sites and one without

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
  - *hidden*: internal, i.e. hidden unit

[3]This number can change after saving, but remains unambiguous. See also chapter 4.2.7.1

  - *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).

- `activation`: The activation value.
- `initial activation` or `i_act`: This variable contains the initial activation value, present after the inital 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 preceeding 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 the unit j.

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

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

This yields the well-known logistic activation function

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

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

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 creates the possibility 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:

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

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 historic reasons. One may think of an f-type as a pointer to some prototype unit where a number of parameters already has 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 it would be possible in a 2D presentation.
- `layers`: Units can be visualized in 2D in up to 8 layers[5], that 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 only selected portions of the network can be shown. 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 *source unit*, or *source* for short, while the other is called *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 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 preceeding 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 3.4). In many models a full connectivity between all units of adjoining levels is assumed.

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

Weights are represented as floats with nine decimals accuracy.

3.1.3 Sites

A unit with sites doesn't have a direct input any more. All incomming links lead to different sites, where the arriving weighted output signals of preceeding 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 in the so called *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 after 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 on.

Additionally, there are 11 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 through 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, that are trained with 'backpropagation through time'.

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 behaviour. This modification is very often based on the Hebb-rule, which states that a link between two units is strengthened, if both units are active at the same time. The Hebb-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 cummulated 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	learing 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 **remote** 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 An Example of a simple Network

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 3.3 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`

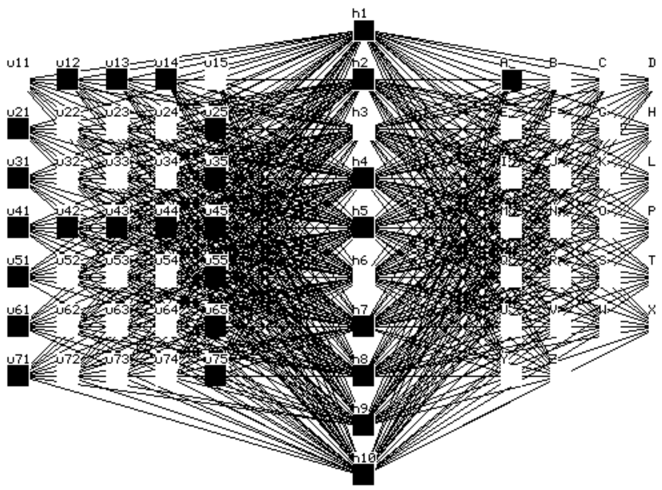

Figure 3.3: Example network of the letter classifier

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 clasifier.

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 two dimensional X-Window 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 next chapter 5 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 4.2.7)
`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 `sparc`, `dec`, `sun3`, `RS6000`, `hp`, `pc386` (with Unix System V) or `other`.

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 4.2.8.

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 8x13 font[1].

`snns -dfont 8x13` 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 **Info** panel called `xgui-info`, near the bottom left the **Menu** button GUI, to open other windows, a message line at the right of this button, and a line with status information at the bottom .
- several **Displays**, to display the network graphically in two dimensions.
- **File** browser for loading and saving networks and pattern files (called with the button GUI in the manager panel).

[1] On some systems the fonts 7x14 or 7x14bold are preferable

- **3D View** panel to control the three dimensional network visualization component.
- **Remote** panel for simulator operations.
- **Bignet** panel to facilitate the creation of big regular feed-forward nets, time delay, ART1, ART2 and ARTMAP networks.
- **Cascade** panel for control of the learning phase of cascade correlation learning.
- **Graph** display, to explain the network error during teaching graphically.
- **Inversion** display, to control the network analyzing tool.
- **Weight Display**, to show the weight matrix as a WV- or Hinton diagram.
- **Help windows** to display the help text.

Of these windows only the Manager panel and one or more 2D displays are open from the start, the other windows are opened with the button `GUI` in the manager panel.

Additionally, there are several popup windows (*transient shells*) which only become visible when called and block all other XGUI windows. Some of them are mentioned below:

- **Setup** panel for adjustments of the graphical representation. (called with the button `SETUP` in the displays)
- **Layer** panel for setting the layer numbers (called with the button `LAYERS` in the Setup panel)
- **Site edit** panel for editing prototypes and sites. (`OPTIONS` in the remote panel)

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. Of the above mentioned popup windows, the file panel is the most important, since it is needed to load or save networks, pattern and configuration files.

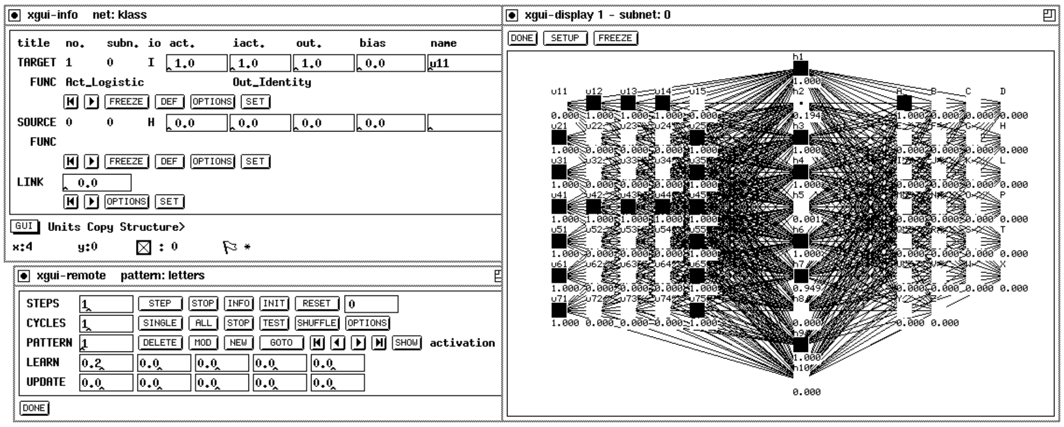

Figure 4.1: Manager panel, remote panel and a display.

Figure 4.1 shows a typical screen setup. The following description explains the tasks and possibilities of the various windows. A detailed description of the handling will be presented later.

The **Manager** panel contains all elements needed for working with the interface. It should therefore always be kept visible. The **Info** panel in the Manager 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.

The other windows are called with the button `GUI`. `QUIT` is used to leave XGUI. The message line shows information about current operations, like additional information on severe errors.

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 5).

The **Remote** panel constitutes a remote control for the simulator operations.

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.

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.

4.2.1 Confirmer

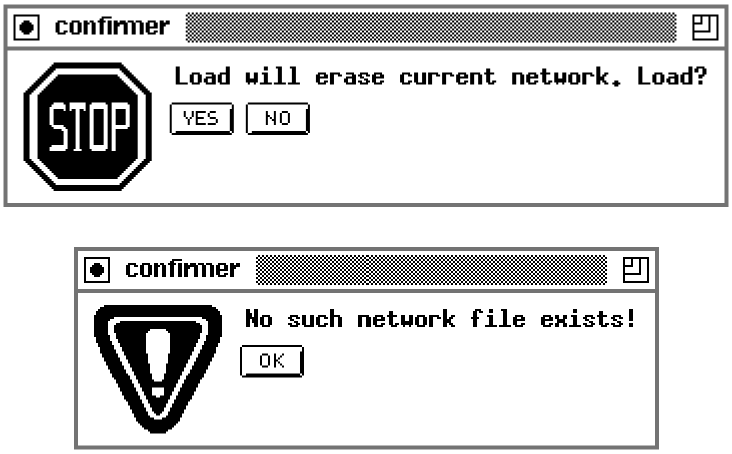

Figure 4.2: A normal confirmer and a message confirmer.

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 4.2).

4.2.2 Manager Panel

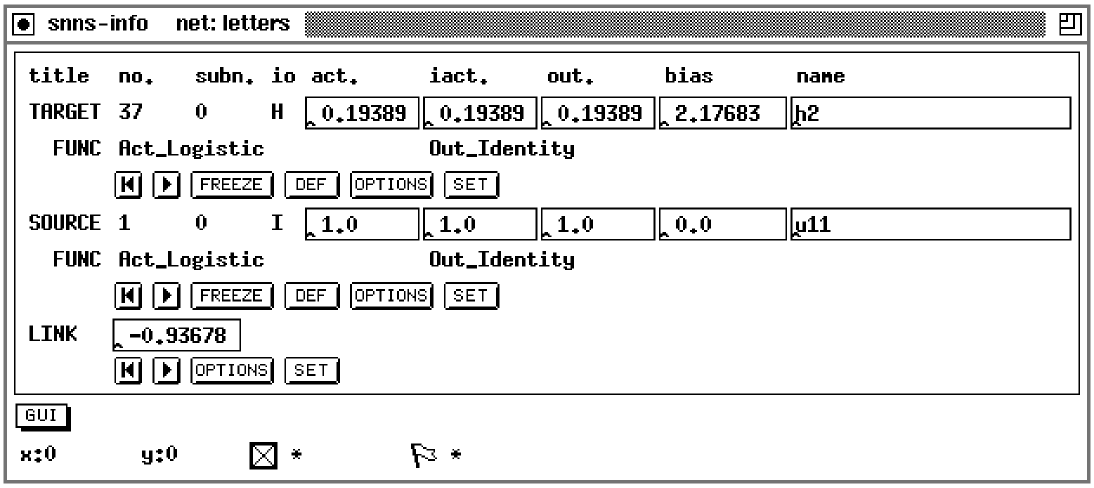

Figure 4.3: Manager panel

Figure 4.3 shows the **Manager** panel with info panel, a message and the status line. 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.

1. Button GUI

   If this menu button is clicked and the mouse button is kept pressed, the following menu to request a window appears. The user can request several displays or help windows, but only one remote panel or text window.

   ```
   FILE
   DISPLAY
   3D VIEW
   INVERSION
   WEIGHTS
   GRAPH
   REMOTE
   BIGNET
   CASCADE
   PRINT
   HELP
   QUIT
   ```

2. Manager Message

   This line features messages about a current operation or its termination. Possible messages are:

`Found ...`	help topic was found.
`Update link done.`	new weight was assigned.
`Listing ...`	a list is being produced.
`n steps. Calculating ...`	n update steps are performed right now.
`Saving ...`	something is being saved.
`...saved.`	...got saved.
`Loading ...`	something is beingloaded.
`...loaded.`	...got loaded.
`HELP: ...`	help information to ...is displayed.

Warnings and errors are also displayed here. The following warnings are possible:

`No target/source unit selected`	action can not be performed, because no target/source is selected in the info panel.
`Can't update link. ...`	weight can not be assigned, because link does not exist.
`This unit has no successors!`	the unit has no successors.
`This unit has no predecessors!`	the unit has no predecessors.
`No more units in this network!`	net has no further units.
`LOAD/SAVE ...aborted.`	load/save was aborted, either by the user, or by an error.

This 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. In most cases, however, only a blinking is visible, because the commands are executed too fast. For a listing of the comand sequences see chapter 5.

3. 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.

   If `safe` appears next to the flag icon, the safety flag was set by the user (see chapter 5). In this case XGUI forces the user to confirm any delete actions.

   The next icon shows a small seleted unit. The corresponding number 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 5 and 5.3).

4.2.3 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→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.

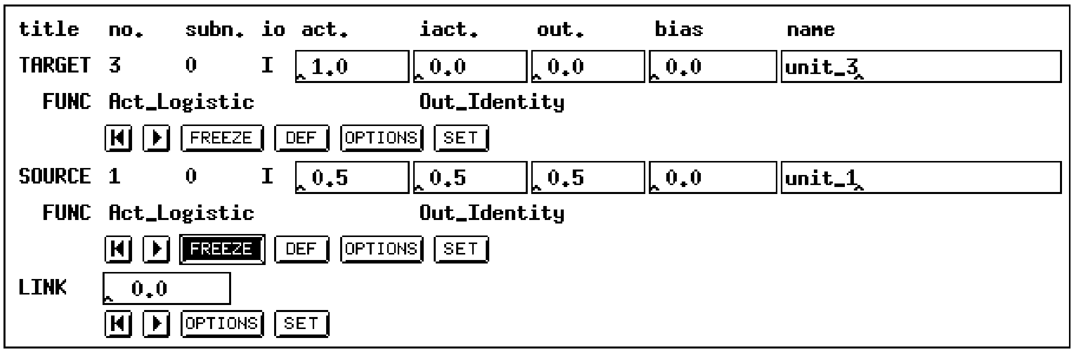

Figure 4.4: Info panel

In table 4.1 the various fields are listed. The fields in the line FUNC have the following meaning (from left to right): Name of the activation function, name of the output function, name of the f-type. The fields in the line LINK have the following meaning: weight, site value, site function, name of the site.

Unit number, unit subnet number, site value, and site function can not be modified.

Note: The specified SNNS value ranges must be obeyed. Values outside the specified range are not rejected by the graphical user interface. Numerical values of the type `float` have the following format: Sign, one digit, decimal point, and five decimal digits. For bias and weight two digits before the decimal point are critical. 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: Select first TARGET of SOURCE (arrow button at TARGET) or select first SOURCE of the TARGET (arrow button at SOURCE).
2. Arrow button: Select next TARGET of SOURCE (arrow button at TARGET) or select next SOURCE of the TARGET (arrow button at SOURCE).
3. FREEZE: Unit is frozen, if this button is inverted. Changes become active only after SET is clicked.

Name	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; usually $-1.0..+1.0$
iact. (initial act.)	Text	input	float; usually $-1.0..+1.0$
out. (output)	Text	input	float; usually $-1.0..+1.0$
bias	Text	input	float
name	Text	input	25 letters or underscore
func (`act_*`)	Label	`OPTIONS`	as available
func (`out_*`)	Label	`OPTIONS`	as available
link (weight)	Text	input	float
site (site value)	Label		float
func (`site_*`)	Label		as available
name (site name)	Label		as available at `TARGET`

Table 4.1: Table of the unit, link and site fields in the Info panel

4. `DEF`: The default unit is assigned the displayed values of `TARGET` and `SOURCE` assigned (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 the current site.
`list all sites of TARGET`	list all sites of the `TARGET`
`list all links from SOURCE`	list all links starting at the `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 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 *scaleFaktor* 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[2], 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.

[2]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!

4.2.5 Setup Panel

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.

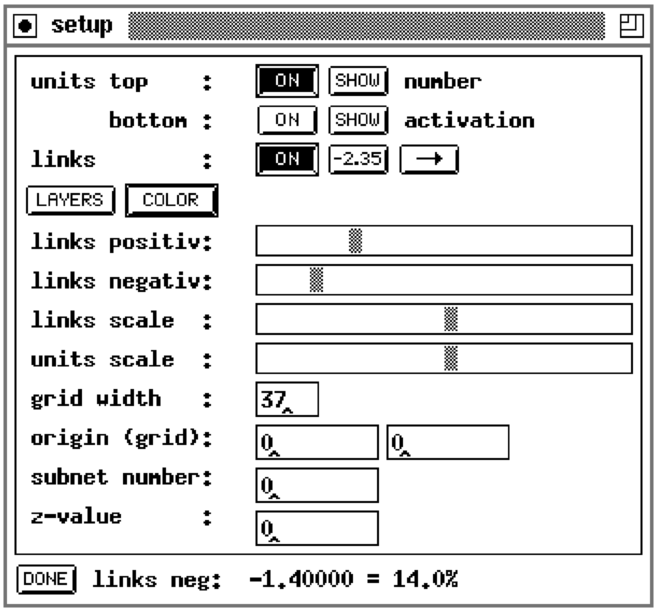

Figure 4.5: Setup Panel.

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. $\boxed{\texttt{LAYERS}}$ 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. $\boxed{\texttt{COLOR}}$ 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:

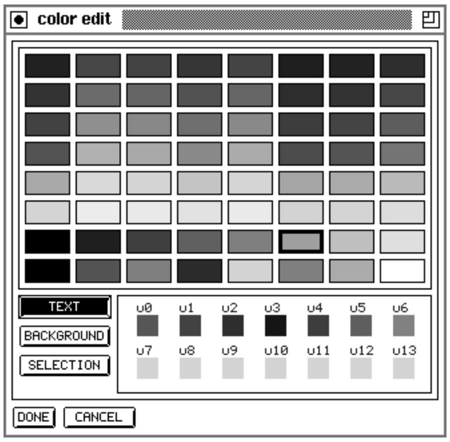

Figure 4.6: Color Setup Panel.

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 ($\boxed{\texttt{TEXT}}$, $\boxed{\texttt{BACKGROUND}}$, or $\boxed{\texttt{SELECTION}}$) 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 $\boxed{\texttt{DONE}}$ button.

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

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 4.2.4). 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 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.

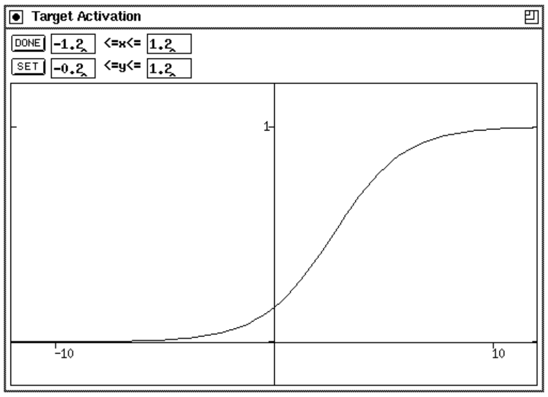

Figure 4.7: The logistic activation function in a new picture range

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 the displayed function changes, e.g. because a new activation 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).

4.2.7 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.

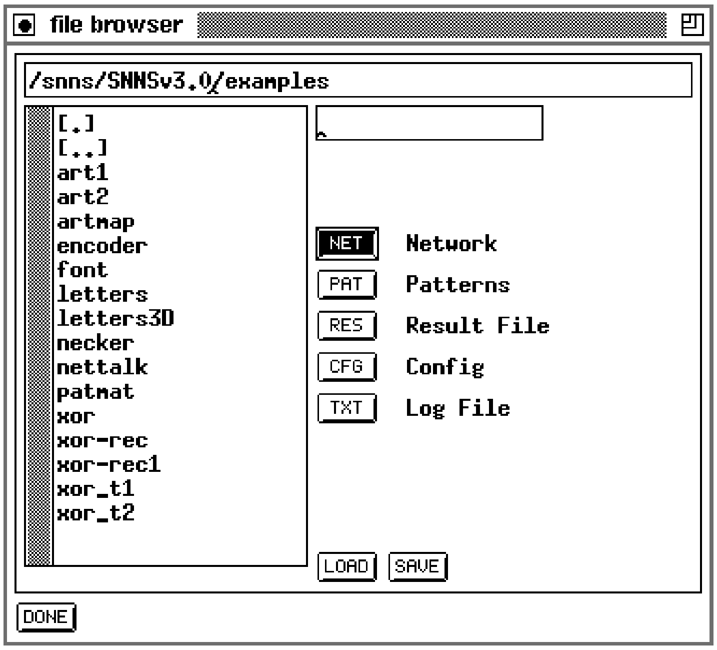

Figure 4.8: File Panel.

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 of the corresponding buttons below. With the setting of picture 4.8 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: In version 3.0 the directories must be executable in order to be processed properly by the program!

4.2.7.1 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.

4.2.7.2 Loading and Saving Patterns

Patterns are combinations of activations of input or output units. Pattern files, like nets, are administrated by the SNNS kernel. During loading the kernel checks whether the number of input and output units is the same as in the network in memory. If this is not the case, the operation is aborted. The filename of the pattern loaded last is displayed in the window title of the remote 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).

4.2.7.3 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.

4.2.7.4 Saving a Result file

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 4.9 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.

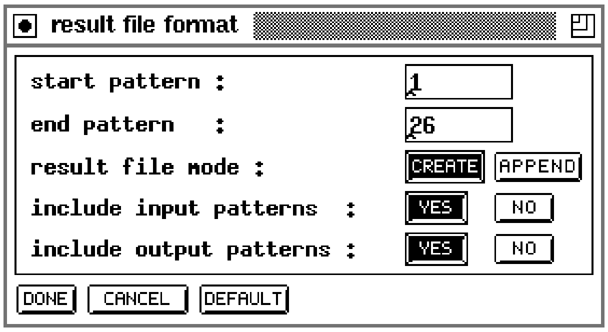

Figure 4.9: Result File Popup

4.2.7.5 Defining the Log File

In the **log file** messages to `stdout` can be stored which document the simulation run. 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`.

4.2.8 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.

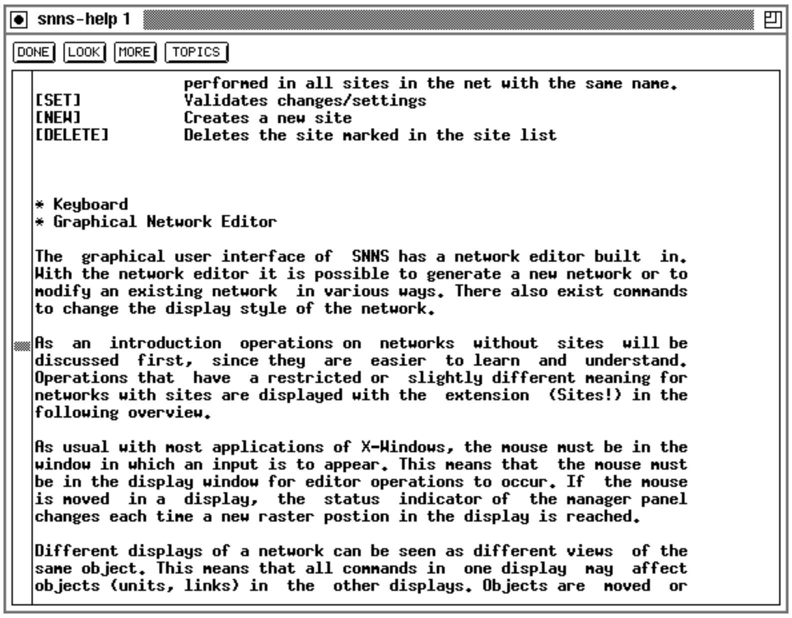

Figure 4.10: Help Window

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`.

4.2.9 Print Panel

The print panel handels 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.

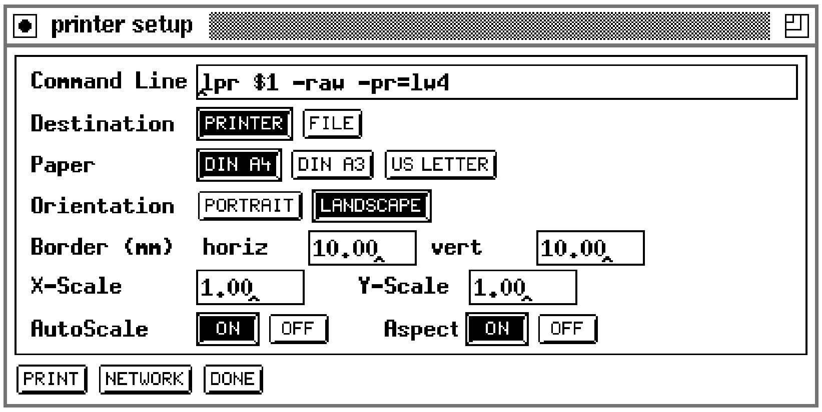

Figure 4.11: Printer panel

The following fields can be set in the Printer Panel, which is shown in figure 4.11.

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 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 maximum possible size on the paper if turned on.
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.

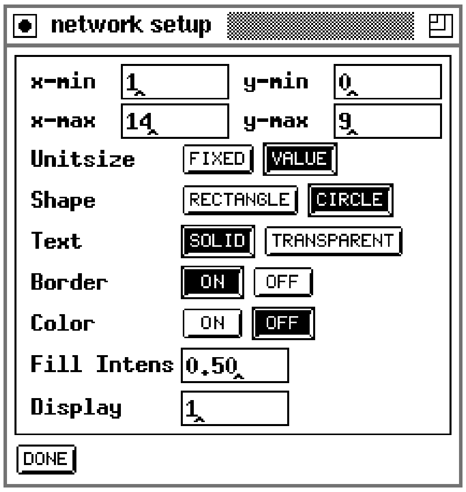

Figure 4.12: The Network Panel

4.2.10 Remote Panel

With this window the simulator is operated (as with a remote control). Figure 4.13 shows this window. Table 4.2 lists all window elements. 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 remote panel.

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

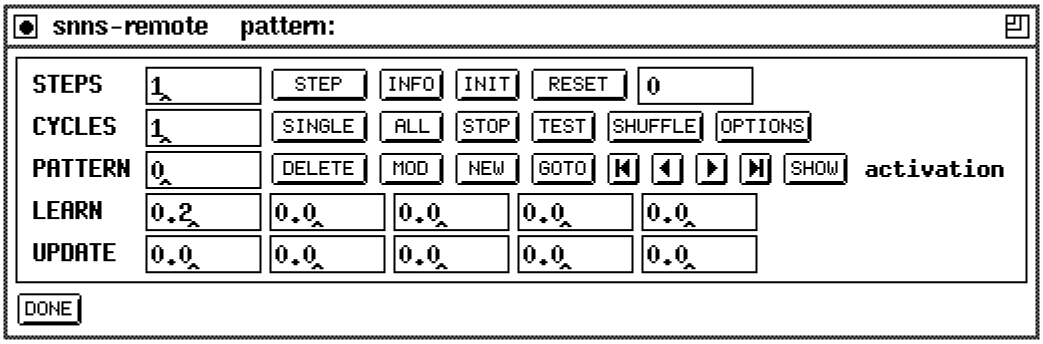

Figure 4.13: Remote Panel

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

Table 4.2: Input fields of the remote panel

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

4. `INIT`: A popup window to set parameters to initialize the network is called.

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

6. The text field after `RESET` displays the steps, executed so far.

7. `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.

8. `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`.

9. `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`.

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

11. `TEST`: With this button, the user can test the behaviour 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.

12. `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.

13. `OPTIONS`: Offers the following menu:

`select update function`	select the order for updating the activations
`select learning function`	select the learning function
`select init function`	select the initialization function
`jog weights`	change all link weights randomly
`edit f-types`	edit/create f-types
`edit sites`	edit/create sites
`delete all patterns`	delete all patterns in main memory
`clear SNNS`	delete network and patterns

   The first six menu items open a popup window for further selections, while the last two take effect immediately.

   If `jog weights` is selected, a popup window appears to specify the range (`low limit .. high limit`). This operation affects all links in the network.

   The menu item `select learning function` 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)
`Backpropagation`	"vanilla" Backpropagation
`BackpropBatch`	Backpropagation for batch training
`BackpropMomentum`	Backpropagation with momentum term
`BackpropBatchThroughTime`	Batch-Backpropagation for recurrent networks
`BackpropThroughTime`	Backpropagation for recurrent networks
`Backpercolation`	Backpercolation 1 (Mark Jurik)
`CascadeCorrelation`	Cascade correlation meta algorithm
`Counterpropagation`	Counterpropagation (Robert Hecht-Nielsen)
`Quickprop`	Quickprop (Scott Fahlman)
`QuickpropThroughTime`	Quickprop for recurrent networks
`RadialBasisLearning`	Radial Basis Functions
`Rec.CascadeCorrelation`	Cascade correlation for recurrent networks
`RProp`	Resilient Propagation learning
`TimeDelayBackprop`	Backpropagation for TDNNs (Alex Waibel)

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

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

16. `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.

17. `NEW`: A new pattern is defined that is added behind existing patterns. Input and output values are defined as above.

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

19. 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.

20. `SHOW`: 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 alternatives:

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

21. `LEARN`: The five parameters of the learning functions vary depending on the learning

functions used. For the learning functions that are already built in into SNNS, they are given in section 4.3.

22. `UPDATE`: The five parameters of the update functions vary depending on the network model used. They are not used in the learning functions distributed in this release of SNNS. They have been used in other network models which we implemented, but currently do not distribute.

4.2.11 Weight Display

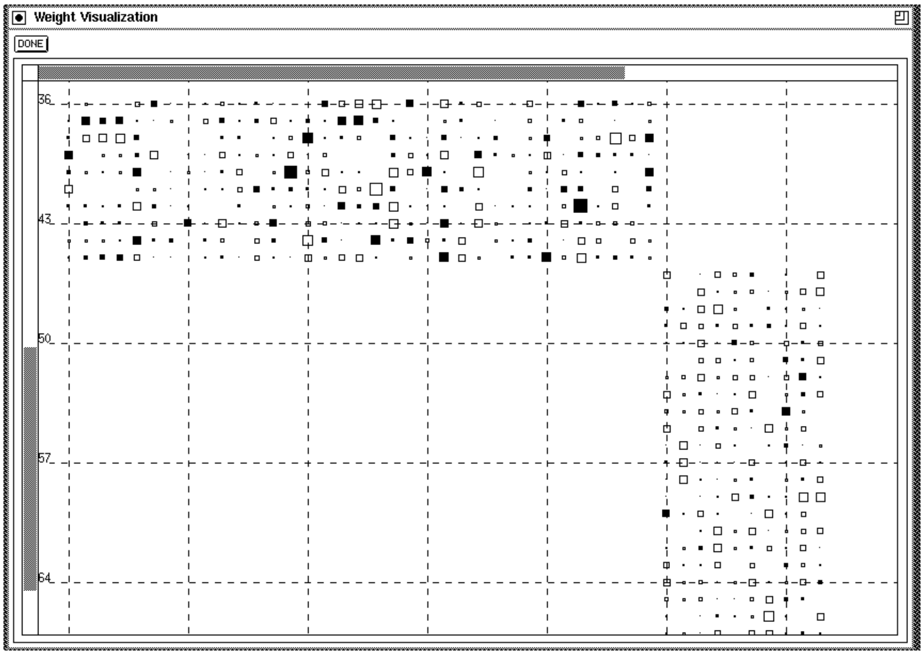

Figure 4.14: A typical Hinton diagram

The weight display window is a separate window specialized for displaying the weights of a network. It is called from the manager panel in the gui menu with the entry **`weights`**. 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 are used (WV-diagrams).

On small networks, all connections are displayed at the same time. With larg nets the display changes to a viewport, where only a small portion of the net is visible and the user is able to move around with scrollbars.

In a Hinton diagram, the size of a square corresponds to the absolute size of the correlated link. A filled square represents negative, an empty square positive 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. The user also has got the possibility to retrieve 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.

4.2.12 Graph Window

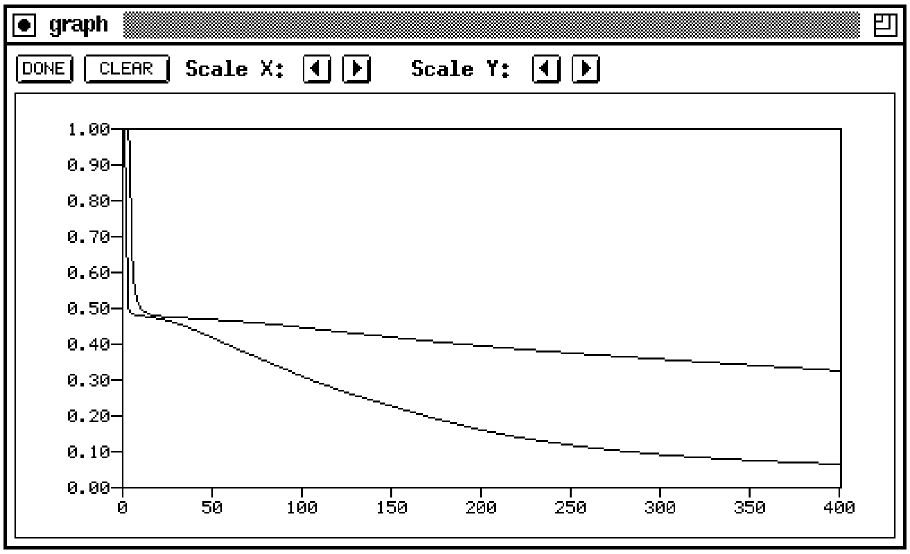

Figure 4.15: Graph window

Graph is a tool to visualize the error developement of a net. The program is started by clicking the menu item 'graph' in the pulldown menu under the $\boxed{\texttt{GUI}}$ button of the info panel. Figure 4.15 shows the window which appears on the screen after selecting the entry `GRAPH`.

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[3]. 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 4.15). 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.

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

In the top region of the graph window, six 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.

: Enlarge scale in one direction.

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.

4.3 Parameters of the Learning Functions

In figure 4.16 the remote panel is displayed again for easier reference. The following learning parameters (from left to right) are used by the learning functions that are already built into SNNS:

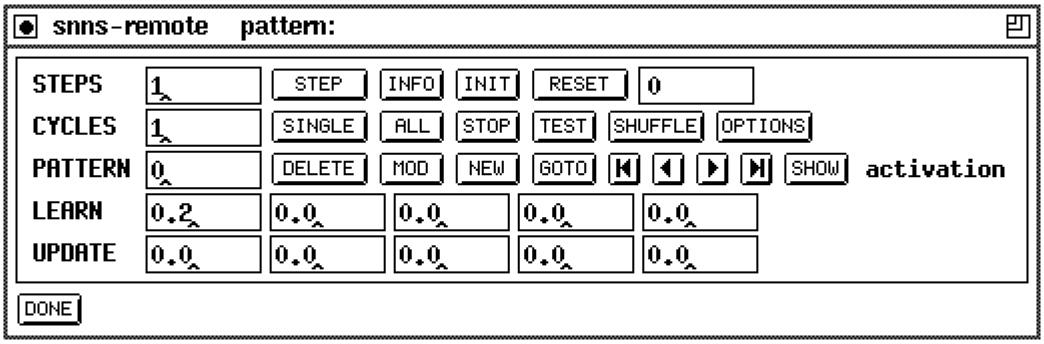

Figure 4.16: Learning parameters in the remote panel

- `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 $0.1 \dots 1.0$. 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 $0 \dots 1.0$.

  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 $0 \dots 0.25$, 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

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

- `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 $0.005 \dots 0.1$.

  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 $0.0 \dots 1.0$.

  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 $0.1 \dots 0.3$.

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.

- `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.

- `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 $\frac{|\mathrm{F}_1^a|}{|\mathrm{F}_0^a|}$)
  2. ρ^b: vigilance parameter for ART^b subnet. (quotient $\frac{|\mathrm{F}_1^b|}{|\mathrm{F}_0^b|}$)
  3. ρ: vigilance parameter for inter ART reset control. (quotient $\frac{|\mathrm{F}^{ab}|}{|\mathrm{F}_2^b|}$)

- `RPROP Learning (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.

- `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:

    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

    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

1. η_1: learning parameter, specifies the step width of the gradient descent when minimizig 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 covarinance. 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 used formula is:

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

4.4 Creating and Editing Unit Prototypes and Sites

Figure 4.17 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.

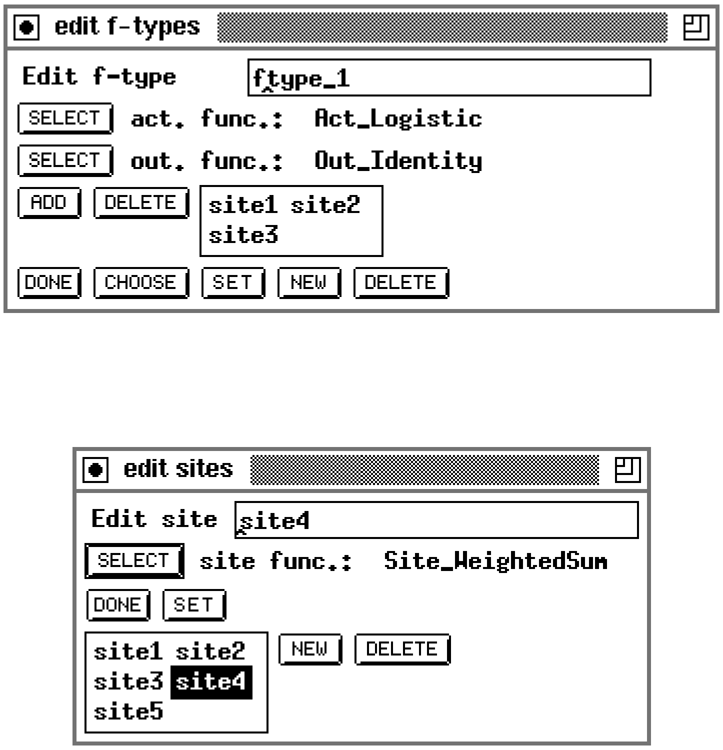

Figure 4.17: Edit panels for unit prototypes (f-types) and sites

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

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 the chapter 5.5.

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 postion 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 out, 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.

5.1 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.

5.2 Selection

5.2.1 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 stylised 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`).

5.2.2 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 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 5.6 a longer example dialogue to build the well known XOR network (see also figure 3.1) is given which shows the main interaction principles.

5.3 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 comand 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. Klicking on a selected unit with the right mouse button only deselects this unit. Klicking 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.

5.4 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 5.1)
- `Help`: get help information. A help window pops up (see chapter 4.2.8)

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 5.5. 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, plus itself *(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 ingoing 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

5.5 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 positon
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.

Connetions 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 5.1. 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)

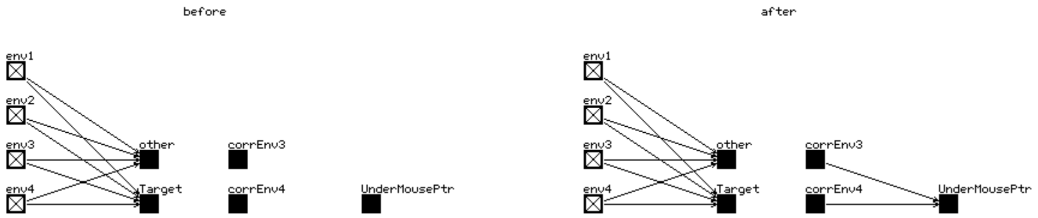

Figure 5.1: Example to `Links Copy Environment`

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 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 recomended 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 postion 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.

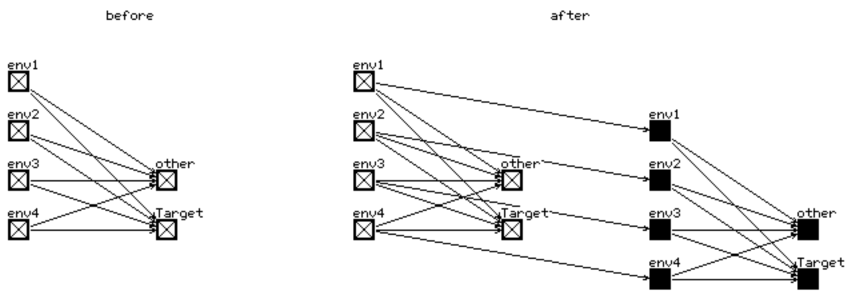

Figure 5.2: An Example for `Units Copy Structure with Forward binding`

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 5.2. 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 usefull after a deletion of links.

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

    This command assigns arrow heads 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.

5.6 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 6

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 menu item BigNet in the pull-down menu, which appears when pressing the GUI button. This invokes the selection menu of figure 6.1, where the individual tools can be selected. This chapter gives a short indroduction to the handling of each of them.

Figure 6.1: The BigNet Menu Window

6.1 BigNet for Feed-Forward Networks

6.1.1 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. 6.3).

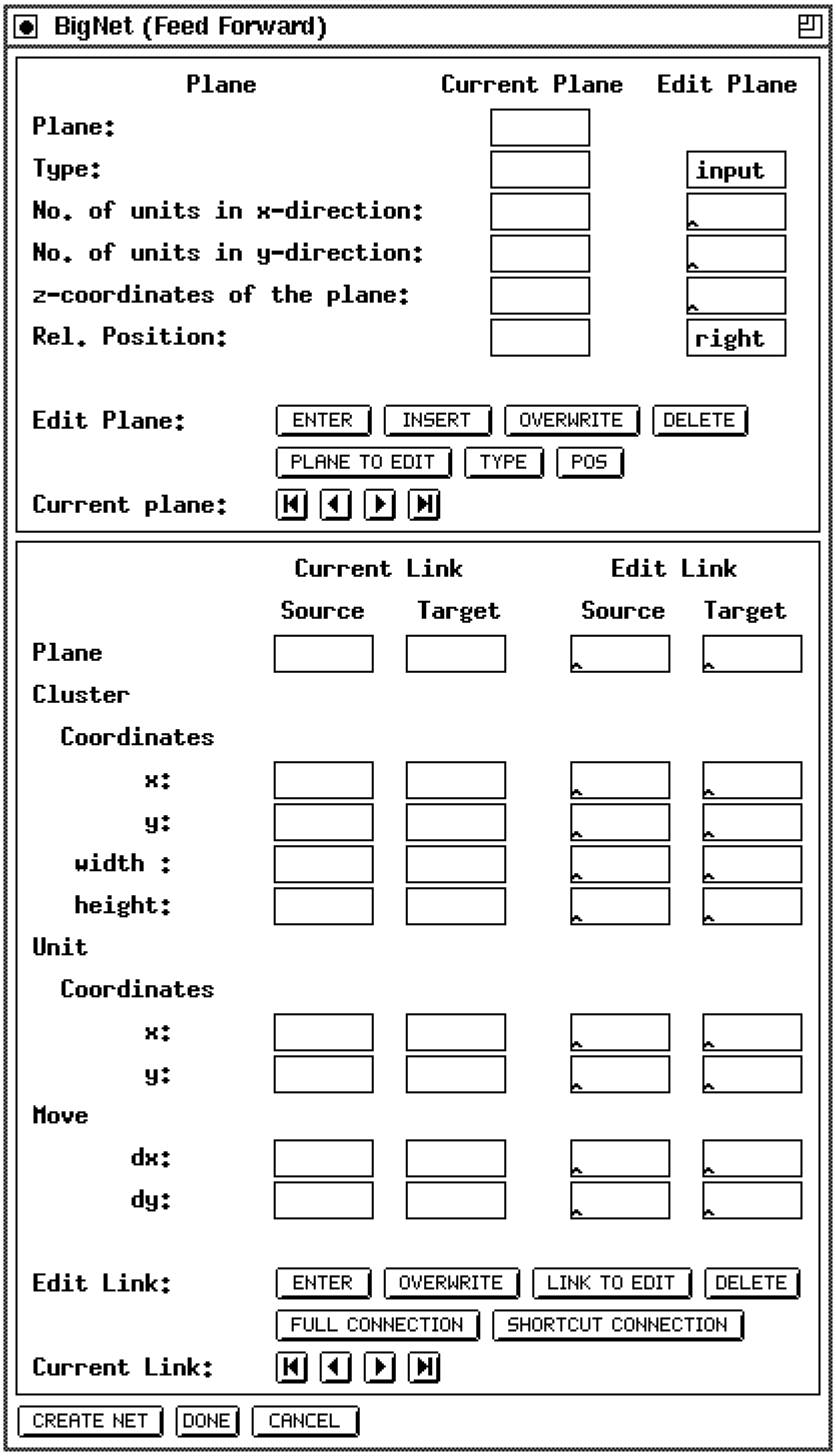

Figure 6.2: The BigNet window for Feed-Forward Networks

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.

6.1.2 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 6.4. BigNet then automatically generates the coordinates of the unis.

`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 $1 \leq i \leq n-1$.

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

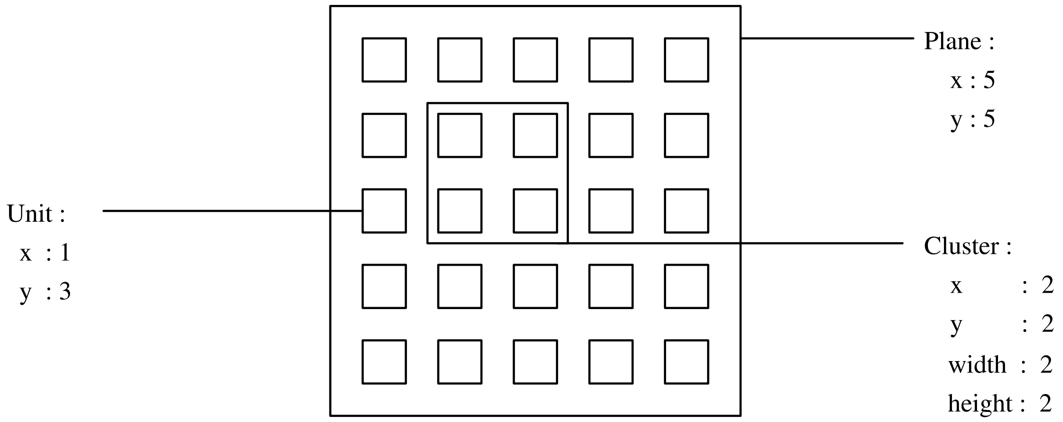

Figure 6.3: Clusters and units in BigNet

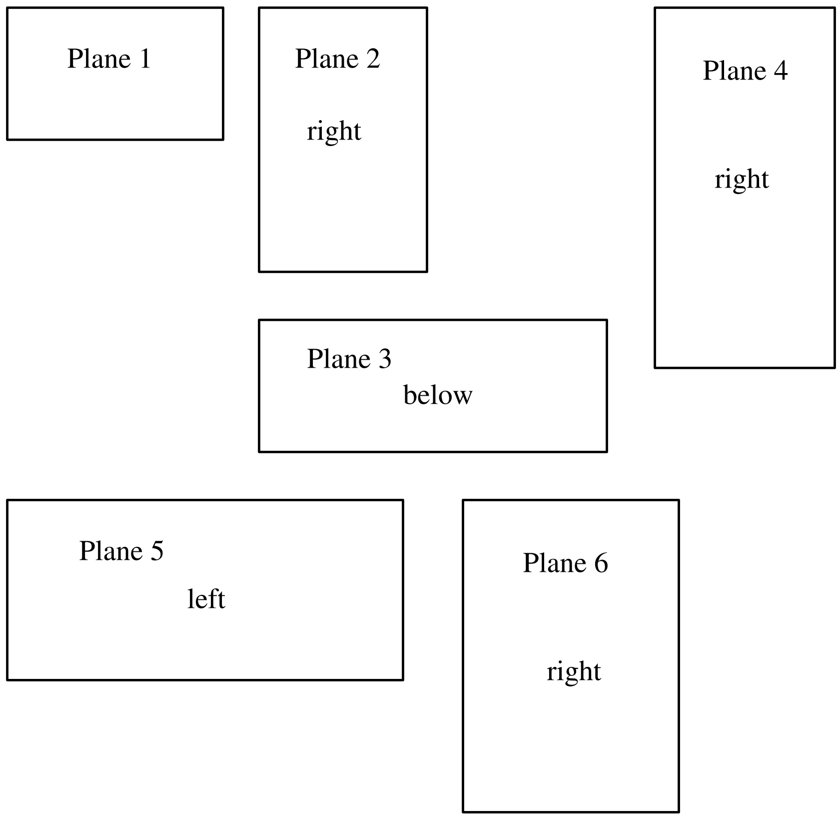

Figure 6.4: Positioning of the planes

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.

6.1.3 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 6.4 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.

6.1.4 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 6.5 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.

	Current-Link		Edit-Link	
	Source	Target	Source	Target
Plane			x	x
Cluster				
x				
y				
width				
height				
Unit				
x				
y				

	Current-Link		Edit-Link	
	Source	Target	Source	Target
Plane			x	x
Cluster				
x				x
y				x
width				x
height				x
Unit				
x				
y				

	Current-Link		Edit-Link	
	Source	Target	Source	Target
Plane			x	x
Cluster				
x				
y				
width				
height				
Unit				
x				x
y				x

Figure 6.5: 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

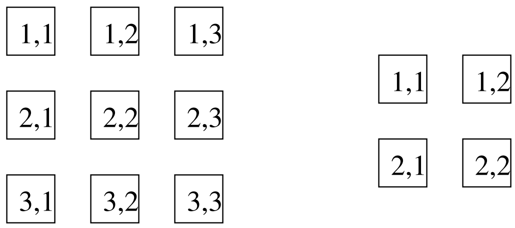

Figure 6.6: The net of example 1

There are two planes given (fig. 6.6). 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 6.7 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 6.7: 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. 6.9). 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 6.8. 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 6.8: Example 2

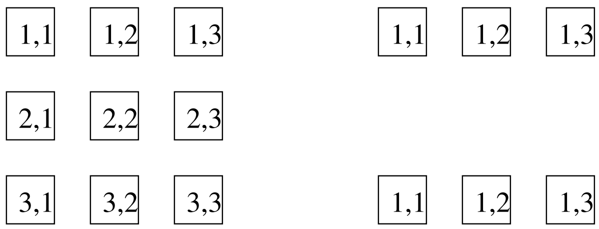

Figure 6.9: The net of example 2

6.1.5 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".

6.2 BigNet for Time-Delay Networks

The BigNet window for Time Delay networks (figure 6.10) 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.

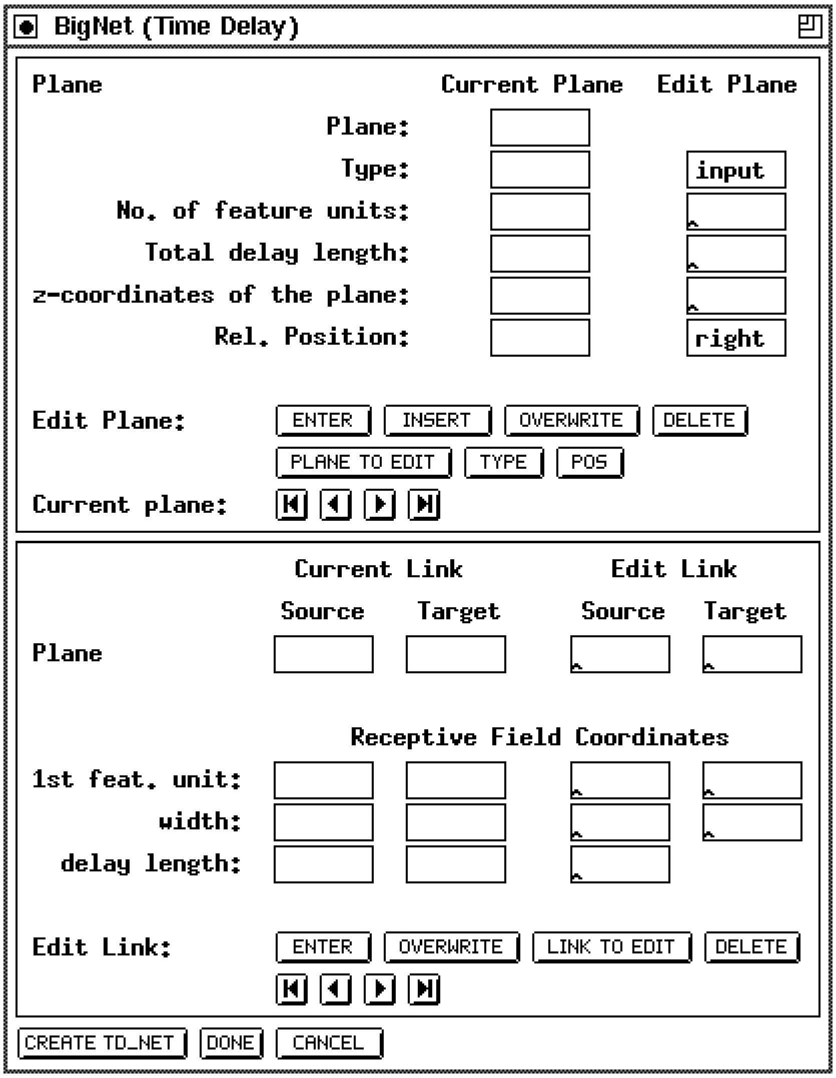

Figure 6.10: The BigNet window for Time Delay Networks

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.

6.2.1 Terminology of Time-Delay BigNet

The following naming conventions have been adopted for the BigNet window. Their meaning may be clarified by figure 6.11.

- 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).

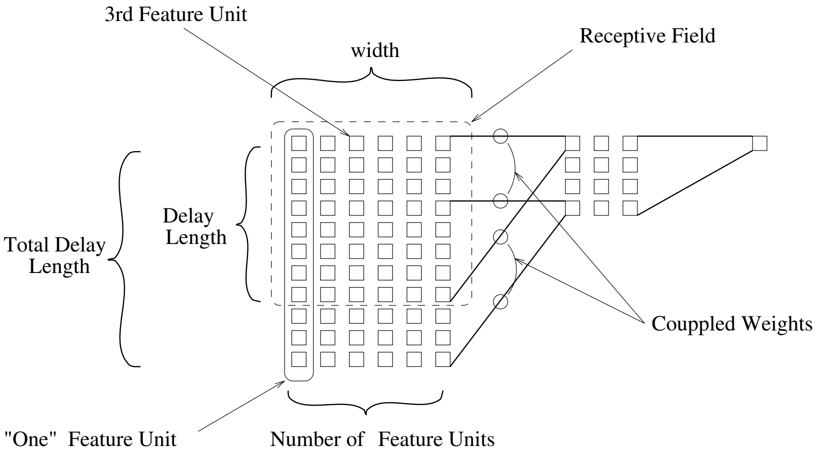

Figure 6.11: The naming conventions

6.2.2 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 6.1.3 for a detailed description.

6.2.3 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 6.12 gives an example of a receptive field specification and the network created thereby.

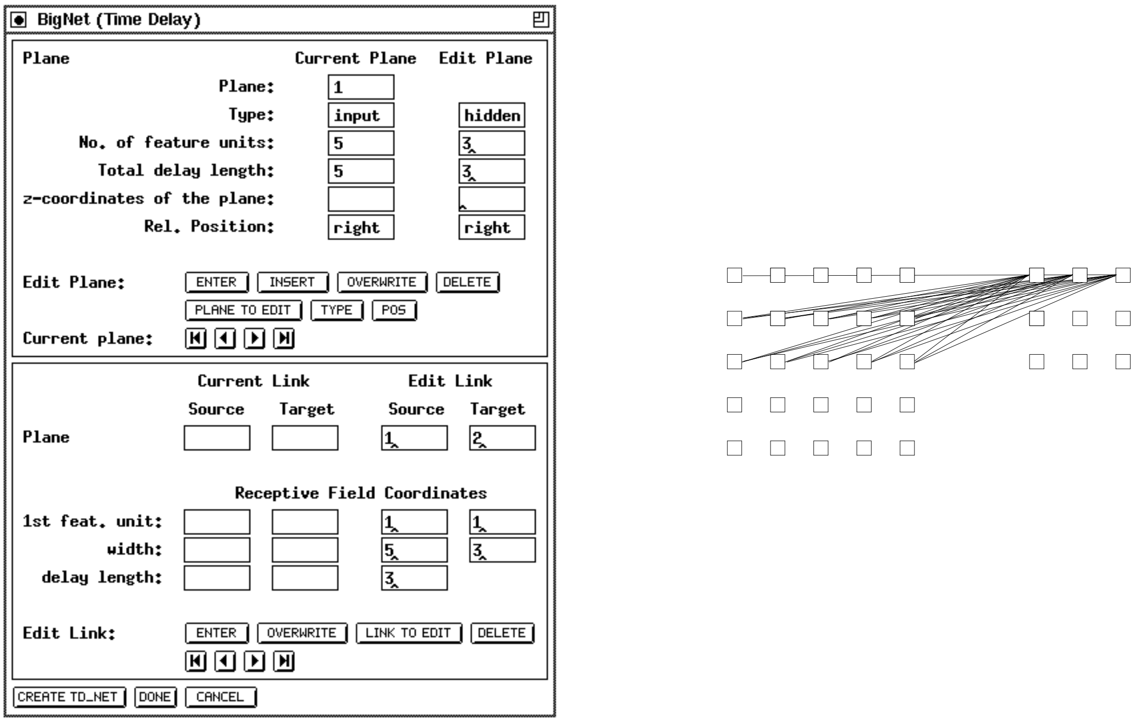

Figure 6.12: An example TDNN construction and the resulting network

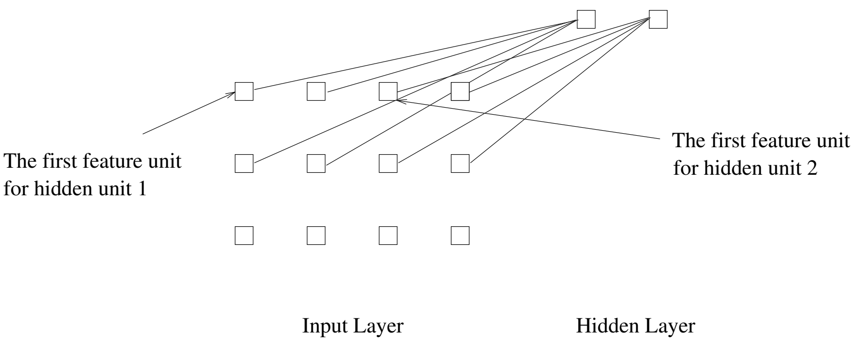

Figure 6.13: Two receptive fields in one layer

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 6.13

6.3 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 6.14 appears on the screen.

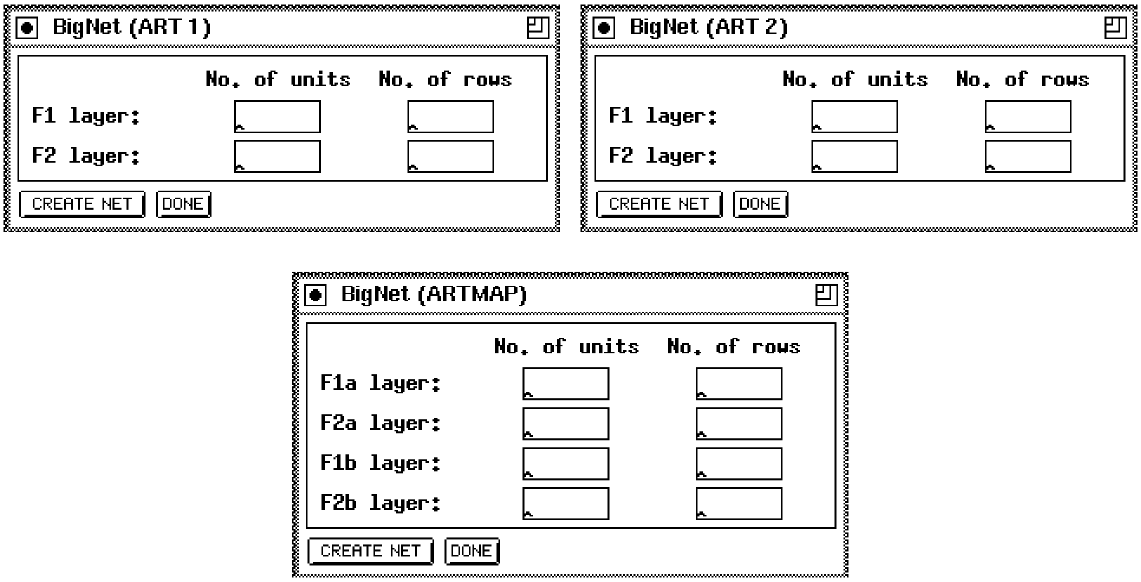

Figure 6.14: The BigNet windows for the ART models

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 6.15.

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

[1]The F_2 layer consists of three internal layers. See chapter 8.11.

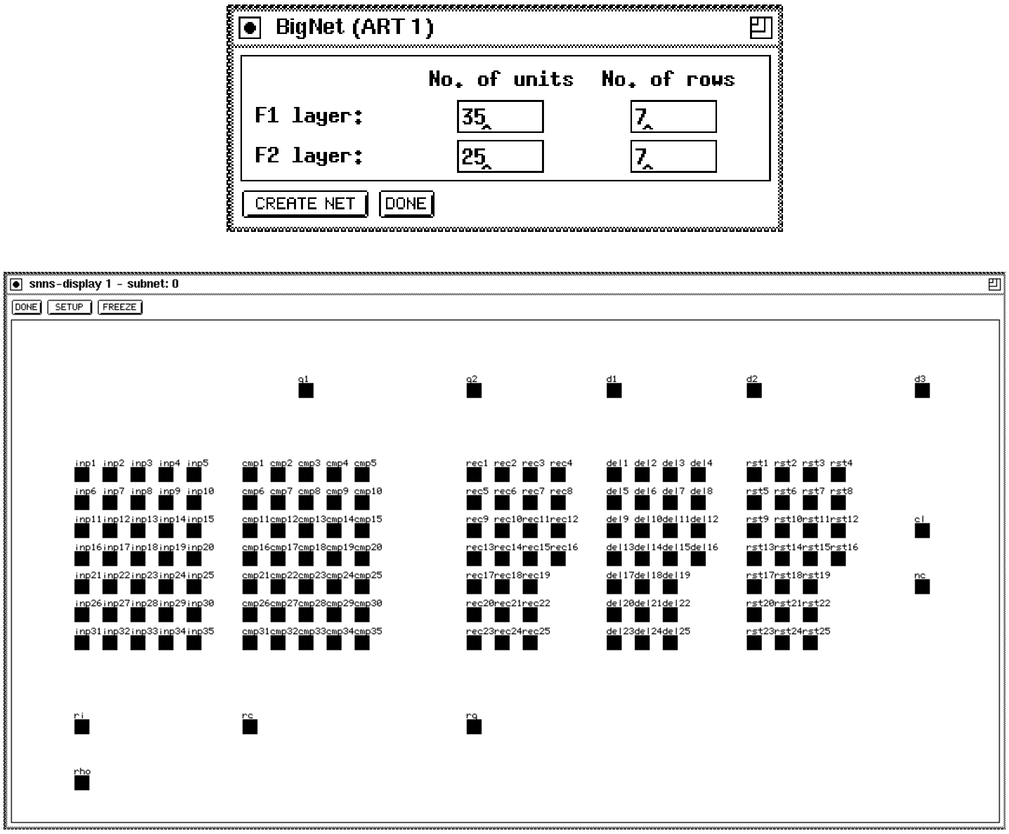

Figure 6.15: Example for the generation of an ART1 network. First the BigNet (ART1) panel is shown with the specified parameters. Next you see the created net as you can see it when using an SNNS display.

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.

Chapter 7

A Network Analyzing Tool

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.

7.1 Inversion 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].

7.2 Inversion Display

The inversion algorithm is called by selecting the menu item `INVERSION` in the pull–down menu hidden behind the `GUI` button of the SNNS Info panel.

Picture 7.1 shows an example of the generated display.

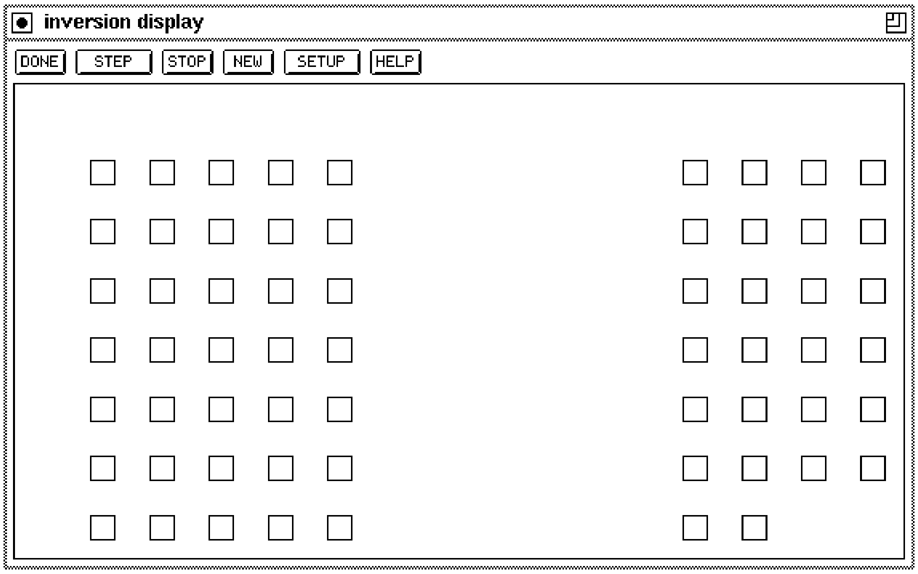

Figure 7.1: The Inversion Display

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. $\boxed{\texttt{DONE}}$: Quits the inversion algorithm and closes the display.

2. $\boxed{\texttt{STEP}}$: 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. $\boxed{\texttt{STOP}}$: 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 $\boxed{\text{STEP}}$ button continues the algorithm at the last position. Alternatively the algorithm can be reset before the restart by a click to the $\boxed{\text{NEW}}$ button, or continued with other parameters after a change in the setup. Since there is no automatic recognition of indefinite loops in the implementation, the $\boxed{\text{STOP}}$ button is also necessary when the algorithm obviously does not converge.

4. $\boxed{\texttt{NEW}}$ 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. $\boxed{\texttt{SETUP}}$: Opens a popup 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 deviation of the activation 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 $\boxed{\text{HELP}}$ 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 instable. An inversion run may converge, while another with only slightly changed variable settings may run indefinitely. The user therefore may have to try several variable combinations 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.

7.3 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 remote 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 7.2 shows an example of a generated input pattern. Here the minimum active units for recognition of the letter 'V' are given. Picture 7.3 shows the corresponding original pattern.

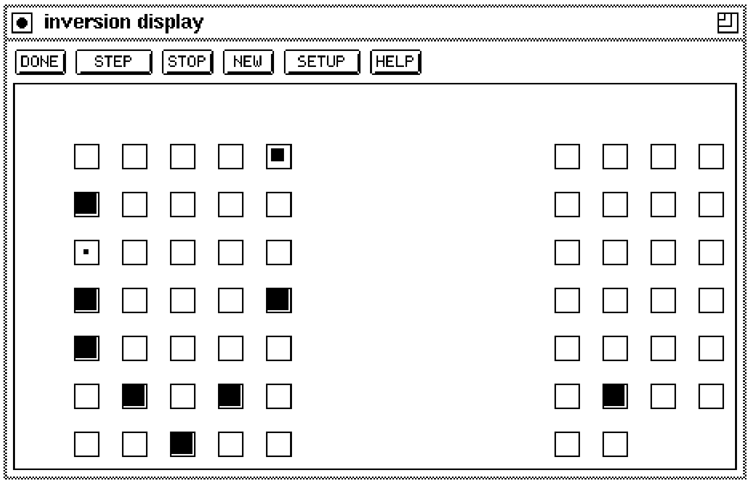

Figure 7.2: An Example of an Inversion Display

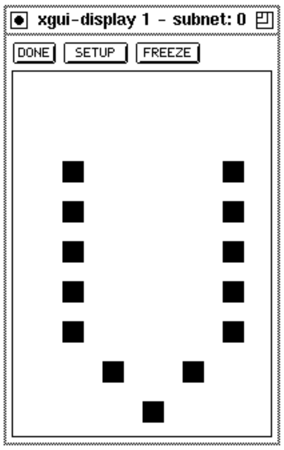

Figure 7.3: The original pattern for the letter V

Chapter 8

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 might not be commonly known. They can not, however, be explained exhaustively here. We refer interested users to the literature.

8.1 Backpropagation Networks

8.1.1 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:

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

This algorithm is also called *online backpropagation* because it updates the weights after every training pattern.

8.1.2 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`.

With flat spot elimination the weights are updated according to the above given rule as usual. Then it is tested whether the error decreased due to that change. If so, the change is performed again. This is reiterated until the error starts to increase. Now the new gradient is computed and learning continues.

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 backprop 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 fast with 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.

8.1.3 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.

8.2 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

8.3 RPROP

RPROP stands for 'resilient propagation' and is a new adaptive learning algorithm that considers the local topology of the error function to change its behavior ([MR92]). Its weight update is based on the so-called 'Manhattan' learning rule:

$$\triangle w_{ij}^{(t)} = \begin{cases} -\triangle & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ +\triangle & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\ 0 & , \text{ else} \end{cases}$$

where $\triangle$, the 'update value', is a problem-dependent constant.

Due to its simplicity, this is a very coarse way to adjust the weights, and so it is not surprising that this method does not work satisfactorily with difficult problems in which it is hard to find an acceptable solution in weight space (e.g. strong nonlinear mappings).

The basic idea for the improvement realized by the RPROP algorithm was to obtain more information about the topology of the error function so that the weight update can be done more appropriately. For each weight we introduce its own 'personal' update value $\triangle_{ij}$, which evolves during the learning process according to its local view of the error function E. So we get a second learning rule for the update values themselves:

$$\triangle_{ij}^{(t)} = \begin{cases} \eta^{+} * \triangle_{ij}^{(t-1)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ \eta^{-} * \triangle_{ij}^{(t-1)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\ \triangle_{ij}^{(t-1)} & , \text{ else} \end{cases}$$

$$\text{where } 0 < \eta^{-} < 1 < \eta^{+}$$

.

Note that the update value is not influenced by the *magnitude* of the derivative, but only by the *sign* of two succeeding derivatives. 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 $\triangle_{ij}$ 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.

Similar adaptation strategies can also be found in former approaches. The essential difference, which turned out to make the Rprop algorithm very powerful and efficient, is the fact that the size of the partial derivative no longer influences the size of the weight step actually taken. The size of the weight step is solely determined by the sequence of the sign of the derivative, which has proven to be a reliable hint about the topology of the local error function (by providing some sort of rough second order information).

The update rule for the weights is the same as above with one exception: if the partial derivative changes sign, the previous update step, leading to a jump over the minimum, is reverted:

$$\triangle w_{ij}^{(t)} = -\triangle w_{ij}^{(t-1)} \text{ , if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} < 0$$

Due to that 'backtracking' weight step, the derivative is supposed to change its sign once again in the following step. In order to avoid a double punishment of the update value, there should be no adaptation of the update value in the succeeding step. In practice this can be done by setting $\frac{\partial E}{\partial w_{ij}}^{(t-1)} := 0$ in the $\triangle_{ij}$ adaptation-rule above.

The update values and the weights are changed every time the whole pattern set has been presented to the network (batch learning).

In multiple experiments we found that choosing $\eta^+ = 1.2$ and $\eta^- = 0.5$ always gave the best results. So both parameters are permanently set to the above values and are not changeable by the user.

At the beginning, all update values $\triangle_{ij}$ are set to an initial value $\triangle_0$. The choice of this parameter is not critical, for it is adapted as learning proceeds (default value is 0.1). The second parameter that can be chosen is the upper limit for the update values, $\triangle_{max}$. The default value of $\triangle_{max}$ is 50.0, and this should work well for most of the problems. However, in very difficult tasks it can be useful to restrict the upper bound of the update values in order to avoid unreasonably large weight steps (e.g. $\triangle_{max} := 0.1$).

As shown in a variety of tests, there is often no need to change the default values of the two parameters $\triangle_0$ and $\triangle_{max}$ at all. [MR92], [MR93]. This is a very favorable property of RPROP, leading to a very robust and easy to use but yet very fast learning algorithm.

8.4 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 seperately 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 magnifing 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.

8.5 Counterpropagation

8.5.1 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

$$w_{ic}(t+1) \quad = \quad w_{ic}(t) + \alpha(o_i - w_{ic}(t))$$

   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

$$w_{ck}(t+1) \quad = \quad w_{ck}(t) + \beta(o_k - w_{ck}(t))$$

   All the other weights remain unchanged.

8.5.2 Counterpropagation Implementation in SNNS

To use counterpropagation in SNNS the following functions and variables have to be selected. The initialization function `CPN_Weights`, 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.

8.6 Dynamic Learning Vector Quantization (DLVQ)

8.6.1 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 $\vec{\mu_i}$. 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 bigger than zero, the direction a vector moves can be influenced.

The DLVQ algorithm works in the following way:

1. Load the training data, norm them 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 out of 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 does not increase any more.
3. Now calculate from the vectors of a class w_A, which are associated with a wrong class w_B a new prototype vector μ_A. Choose for every class one of the new mean vectors and add it to the net. Go back to step 2.

8.6.2 DLVQ in SNNS

To start the learning rule 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 the learning rules CC and RCC the text field `CYCLE` in the remote 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 third parameter in the remote panel (see below).

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 topologiy of a net fits to the DLVQ architecture SNNS will order the units and layers indepentently in the following way: From left to right an input layer, a hidden layer and an output layer. The hidden layer itself is ordered by classes.

[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.

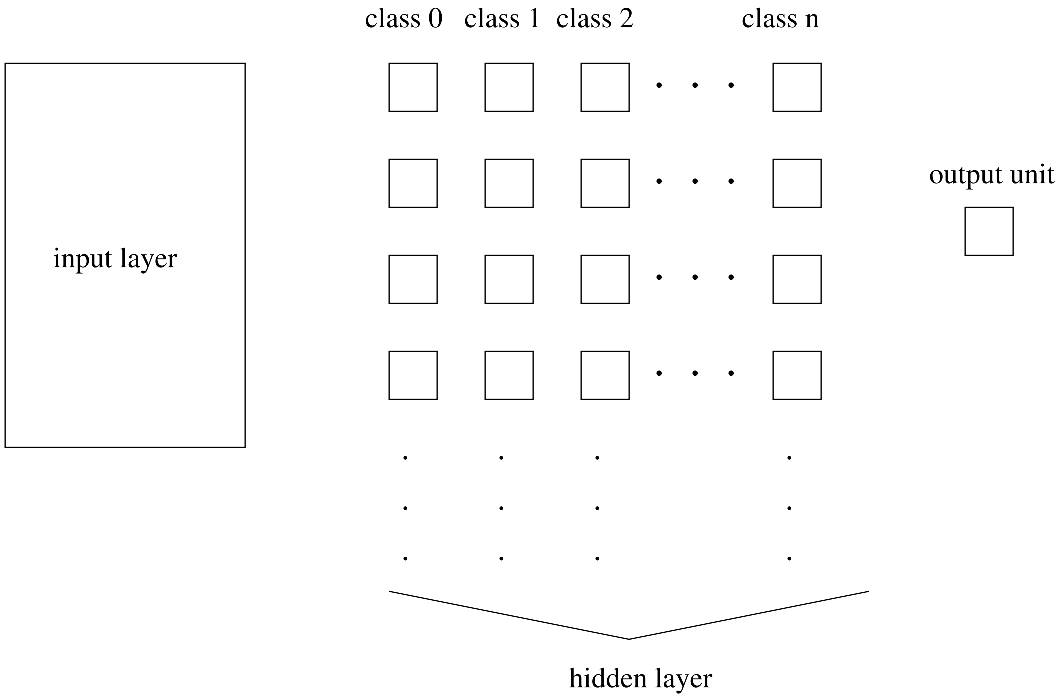

Figure 8.1: Topology of a net which was trained with DLVQ.

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 the information 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 8.1 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 buttom.

8.6.3 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 fits best to classification problems like recognition of patterns, digits and so on. With this algorithm I succeeded in learning 10000 digits with a resolution of 16×16 pixels. Overall the algorithm generated 49 hidden units during learning.

8.7 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 avaliable:

BPTT: Backpropagation through time with online-update.
The gradient for each weight is summed over `backstep` copies between successive 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

[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.

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 hierachical 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 illustating 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].

[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.

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 remote 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 preceeding the reset pattern.

8.8 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 algorihgtms.

8.8.1 Cascade-Correlation (CC)

8.8.1.1 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 a output unit with a usual learning algorithm until the error of the net does not shrink any more.

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 leeding to a candidate unit. Learning takes place with an ordinary learning algorithm. The training is stopped, when the correlation scores does not improve any more.

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 leeding to the new hidden unit are froozen, 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 8.2 shows a net after 3 hidden units have been added.

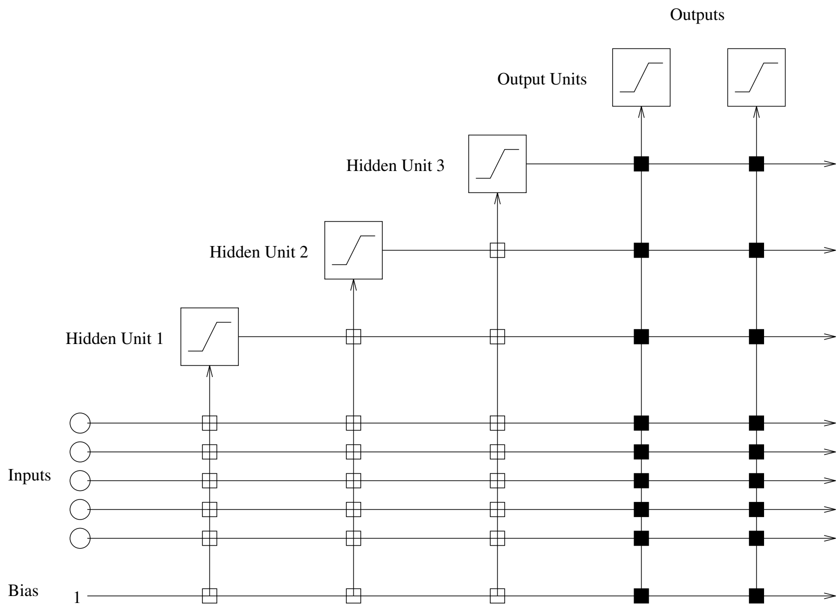

Figure 8.2: A neural net trained with cascade-correlation after 3 hidden units have been added. The vertical lines add all incoming activations. Connections with white boxes are frozen. The black connections are trained repeatedly.

8.8.1.2 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

$$\begin{aligned} e_{po} &= (y_{po} - t_{po}) f_p'(net_o) \\ \frac{\partial E}{\partial w_{io}} &= \sum_p e_{po} I_{ip}, \end{aligned}$$

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 maximation of C proceeds by gradient ascent using

$$\begin{aligned} \delta_p &= \sum_o \sigma_o (e_{po} - \bar{e_j}) f_p' \\ \frac{\partial C}{\partial w_i} &= \sum_p \delta_p I_{pi}, \end{aligned}$$

where σ_o is the sign of the correlation between the candidate unit's output and the residual error at output o.

8.8.2 Recurrent Cascade-Correlation (RCC)

8.8.2.1 The Algorithm

Recurrent Cascade-Correlation (RCC) is a recurrent version of Cascade-Correlation and can be used to train recurrent neural nets ([Elm89]).

Recurrent nets have some features that distinguish them from normal neural networks. For example they can be used to represent time implicitly by its effects on processing 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 networks 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. 8.3). To integrate the Elman model into the cascade architecture some changes are necessery: 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 8.4. 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.

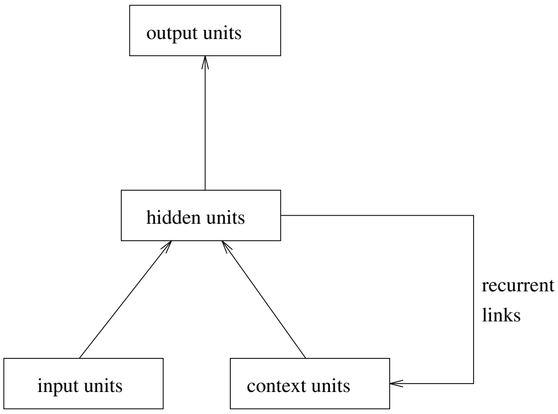

Figure 8.3: The Elman architecture of a recurrent neural net

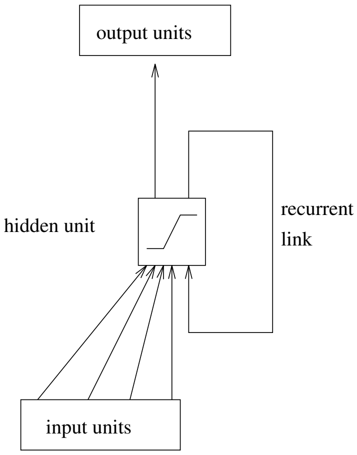

Figure 8.4: RCC architecture of a recurrent neural net.

8.8.2.2 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.

8.8.3 Using the Cascade Algorithms in SNNS

Networks that make use of the cascade correlation architecture can be created in SNNS like all other network types. The control of the training phase, however, is moved from the remote panel to the special cascade window described below. The remote 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 in the following way: From left to right an input layer, a hidden layer, an output layer and a candidate layer [4]. The hidden layer is generated with always 5 units having the same x-coordinate (i.e. above each other on the display).

The cascade correlation control panel, the cascade window (see fig. 8.5, is opened by clicking the menue item `Cascade` in the pull-down menu, which appears when pressing the XGUI button. The cascade window is needed to set the parameters of the learning algorithms CC and RCC. To start Cascade-Correlation, the learning function `CC`, the update function `CC_Update` and the init function `CC_Weights` in the corresponding menues 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.

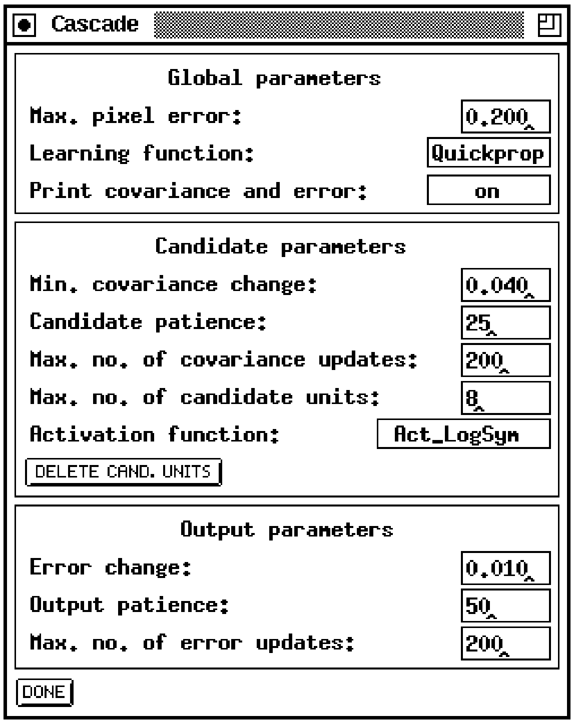

Figure 8.5: The cascade window

- `Global parameters:`
  - `Max. pixel 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.
- `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 progam 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 differnt 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 assignes one of the other activation functions to each candidate unit. The function `Act_LocSym` is identical to `Act_Logistic`, exept 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 error updates:`
    analogous to `Max. no. of covariance updates`

The button DELETE CAND. UNITS deletes all candidate units. In SNNS the candidate units are realized as special units.

8.9 Time Delay Networks (TDNNs)

8.9.1 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 8.6)

- **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*.

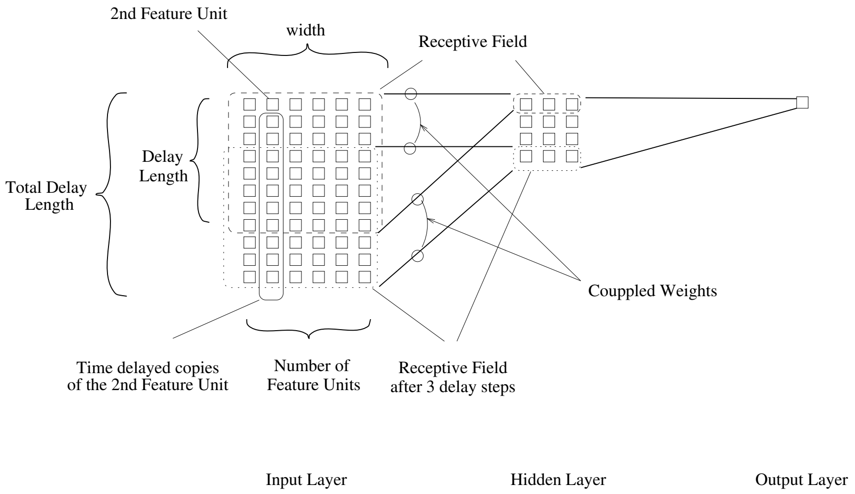

Figure 8.6: The naming conventions of TDNNs

- **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 seperately. Also the units' bias which realizes a spezial sort of link weight is duplicated over all delay steps of a current feature unit. In figure 8.6 only two pairs of coupled links are depicted (out of 54 quadrupels) 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 8.6 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 networks 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 subpatterns. This important feature of TDNNs makes them independent from error-prone preprocessing algorithms for time alignment. The drawback is of course a rather long, since computational intensive, learning phase.

8.9.2 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 physikal 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 standart backpropagation.

8.9.3 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 $\boxed{\text{TEST}}$ and $\boxed{\text{STEP}}$ 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.

8.10 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.

8.10.1 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:

$$\vec{c} = (G^T \cdot G + \lambda G_\square)^{-1} \cdot G^T \cdot \vec{y}$$

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 underdetermined 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 8.7 shows the architecture of the special form of hidden units.

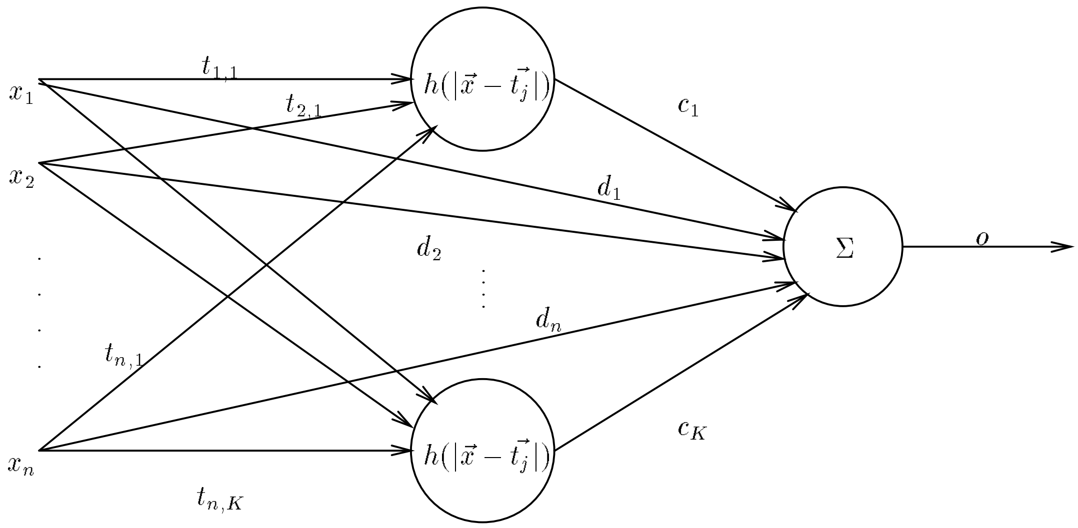

Figure 8.7: The special radial basis unit

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's 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 overtrained, since errors that are smaller than the predefined value are treated as zero. This in turn prevents the corresponding links from being changed.

8.10.2 RBF Implemetation in SNNS

8.10.2.1 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.

8.10.2.2 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 evently 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*, see figure 8.8) 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

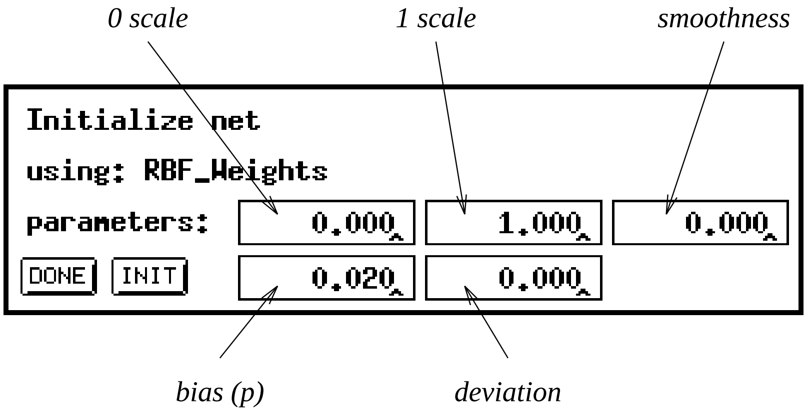

Figure 8.8: complete initialization of all parameters

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:

$$\vec{c} = (G^T \cdot G + \lambda G_{\Box})^{-1} \cdot G^T \cdot \vec{y}$$

The initialization parameter *smoothness* (see figure 8.8) 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 two initialization parameters *0_scale* and *1_ scale* are a remedy for this dilemma (see figure 8.8). 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 like in figure 8.8. 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.

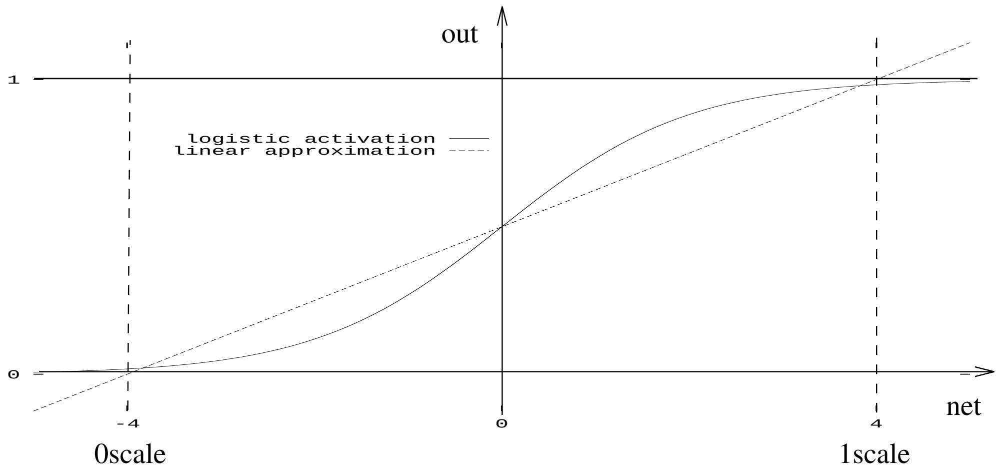

Figure 8.9: Relation between teaching output, input value and logistic activation

Figure 8.9 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 maped 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:

$$\vec{z}_{\mathrm{new}} = \vec{z}_{\mathrm{old}} + \alpha(\vec{l} - \vec{z}_{\mathrm{old}})$$

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.

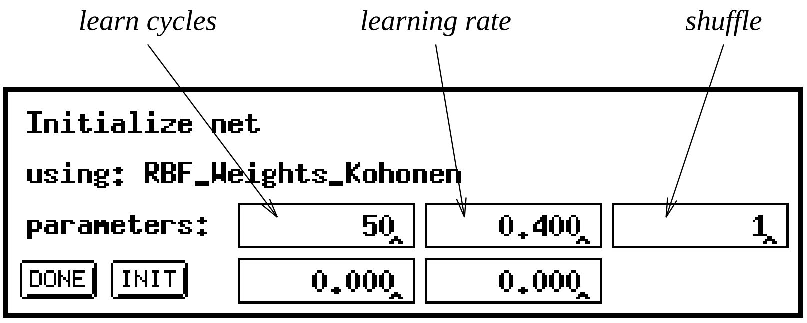

Figure 8.10: Initialization of center vectors

Figure 8.10 shows the meaning and location of the initialization parameters in the popup window.

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.

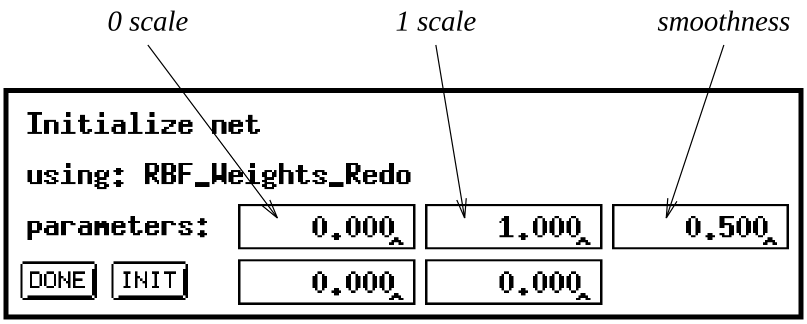

Figure 8.11: Initialization of the second link layer

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`.

8.10.2.3 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`. Figure 8.12 shows the names and the arrangement of the learning parameters:

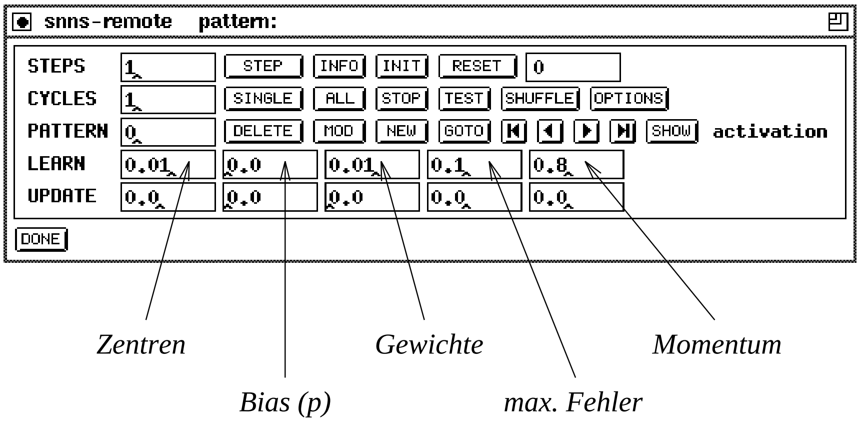

Figure 8.12: Meaning of the learning parameters

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.

8.10.3 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 remote 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 113). When `Act_IdentityPlus-Bias` 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 8.9). 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 8.1). 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 meaningfull 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.

8.11 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 6.

8.11.1 ART1

8.11.1.1 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 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 8.13 the units and links of ART1 networks in SNNS are displayed.

The F_0 or input layer (labeled **inp** in figure 8.13) 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[5]. 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.

[5] This is only important for the chosen realization of the ART1 learning algorithm in SNNS

Figure 8.13: 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.

For an exact definition of the required topology for ART1 networks in SNNS see section 8.11.4

8.11.1.2 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.

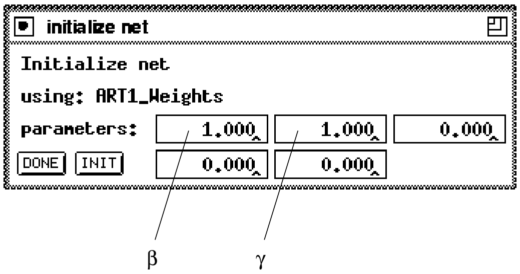

Figure 8.14: ART1 initialization parameters β and γ.

`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

$$0 < \alpha_M < \alpha_{M-1} < \ldots < \alpha_1 \leq \frac{1}{\beta + N}. \tag{8.1}$$

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

$$\alpha_j \equiv \frac{1}{\beta + (1 + j\eta)N}. \tag{8.2}$$

η is calculated out of a new parameter γ and the number of F_2 units M:

$$\eta \equiv \frac{\gamma}{M}. \tag{8.3}$$

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 (figure 8.14). 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 behaviour 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.

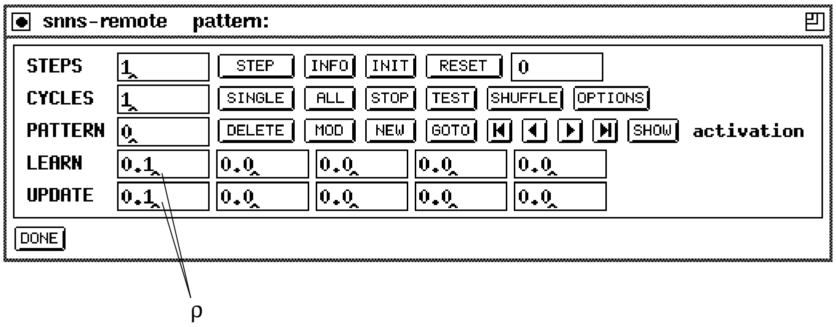

Figure 8.15: Setting the ART1 learning parameter ρ.

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 as shown in figure 8.15 and the number of learning cycles. 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 remote 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 remote 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.

Figure 8.16: 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.

8.11.2 ART2

8.11.2.1 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 8.16 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] and the realisation 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 8.11.4

8.11.2.2 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 8.4 is given by the theory:

$$||\mathbf{z}^J|| \rightarrow \left|\left|\frac{\mathbf{u}}{1-d}\right|\right| = \frac{1}{1-d}, \tag{8.4}$$

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:

$$z_{ij}(0) \leq \frac{1}{(1-d)\sqrt{N}} \qquad \forall\, 1 \leq i \leq N, 1 \leq j \leq M. \tag{8.5}$$

If equality is choosen in equation 8.5, then ART2 will be as sensitive as possible.

To transform the inequality 8.5 to an equation, in order to compute values, we introduce another parameter γ and get:

$$z_{ij}(0) = \frac{1}{\gamma(1-d)\sqrt{N}} \qquad \forall\, 1 \leq i \leq N, 1 \leq j \leq M, \tag{8.6}$$

where $\gamma \geq 1$.

To initialize an ART2 network, you have to select the function `ART2_Weights`. You have to pass the parameters d and γ to the initialization function in the given order, as shown in figure 8.17. (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 behaviour 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.

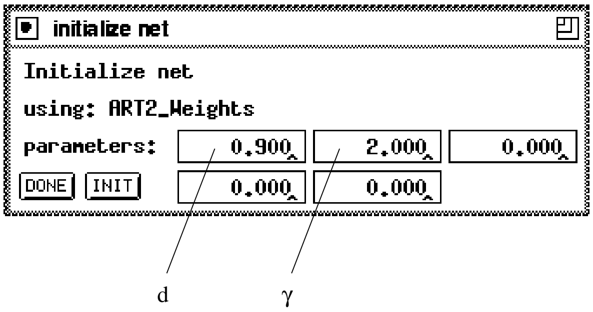

Figure 8.17: ART2 initialization parameters d and γ.

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 choosen to assure quick stabilisation 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 stabilsation c should be choosen 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 continously 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 choosen the learning function `ART2`. As a first step initialize the network with the initialization function `ART2_Weights` described above. Then choose the five parameters ρ, a, b, c and Θ, as shown in figure 8.18. 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.

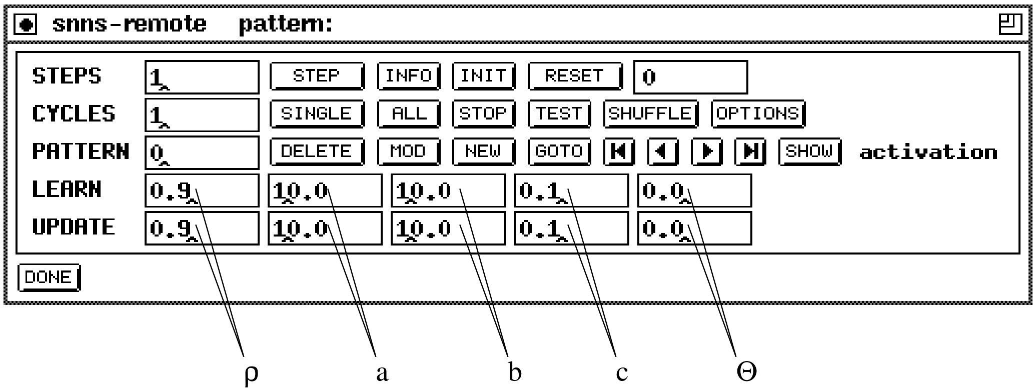

Figure 8.18: Setting the ART2 learning parameters ρ, a, b, c and Θ.

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 remote panel.

8.11.3 ARTMAP

8.11.3.1 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 8.11.1, 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 8.19. 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.

The map field units represent the categories, onto which the ART^a classes are mapped[6]. 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*.

[6]Different ART^a classes may be mapped onto the same category.

Figure 8.19: The MAP field with its control units.

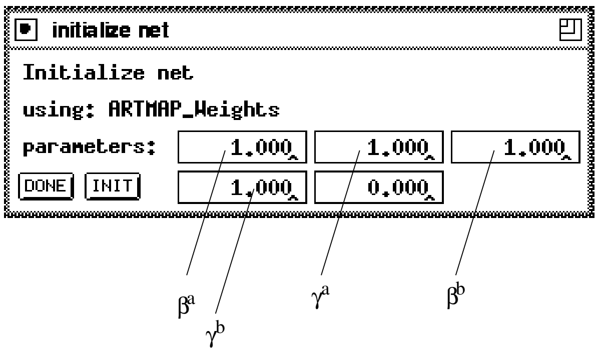

Figure 8.20: ARTMAP initialization parameters β^a and γ^a, β^b and γ^b.

8.11.3.2 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 choose four parameters: β^a, γ^a, β^b and γ^b, as shown in figure 8.20. You can look up the meaning of each pair $\beta^?$, $\gamma^?$ in section 8.11.1.2, 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 ρ, as it is shown in figure 8.21. $\overline{\rho^a}$ is the initial vigilance parameter for the ART^a-part of the net, which may be modiefied 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.

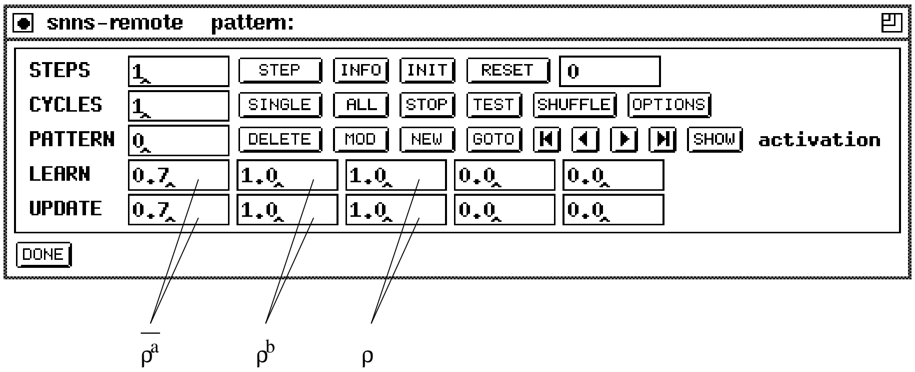

Figure 8.21: Setting the ARTMAP learning parameters $\overline{\rho^a}$, ρ^b and ρ.

ARTMAP Update Functions For ARTMAP two update functions have been implemented, as well:

- `ARTMAP_Stable`
- `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 remote panel. The usage is the same as it is for ART1 and ART2 networks.

8.11.4 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.

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**$_i$ $\mathbf{g}_1$ **ri** $\mathbf{g}_2$	 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**$_j$ $\mathbf{g}_1$	 rec_g1
del$_j$	h	Act_at_least_2	Out_Identity		**cmp**$_i$ $\forall i$ $\mathbf{d}_1$ **rst**$_j$	 rst_signal
$\mathbf{d}_1$	h	Act_at_least_1	Out_Identity		$\mathbf{d}_2$	
$\mathbf{d}_2$	h	Act_at_least_1	Out_Identity		$\mathbf{d}_3$	
$\mathbf{d}_3$	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			
$\mathbf{g}_1$	h	Act_at_least_2	Out_Identity	inp_g1 rec_g1	**cmp**$_i$ $\forall i$	
$\mathbf{g}_2$	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

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**	

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`.

Chapter 9

3D-Visualization of Neural Networks

9.1 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 remote window 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 wireframe 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.

Figure 9.1 shows the topology of a letter classifier as a wireframe model in central projection. In figure 9.2 the hidden lines have been removed and the units are illuminated.

Figure 9.3 shows the activation of the units in the network. Since the network is already trained, unit A shows maximum activation. The other output units are not activated and are therefore not shown in the 3D-display.

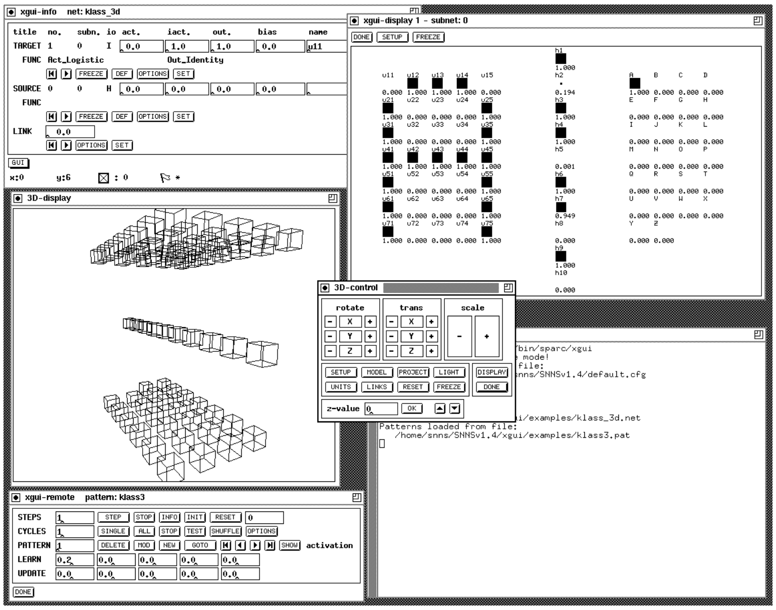

Figure 9.1: Topology of a network as wire frame model without links

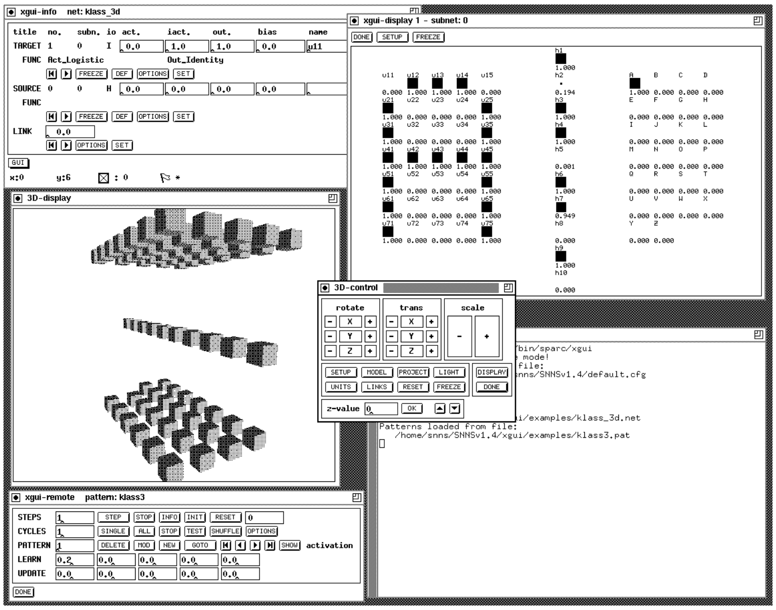

Figure 9.2: Topology of a network as solid model without links

9.2 Use of the 3D-Interface

9.2.1 Structure of the 3D-Interface

The 3D interface consists of three parts:

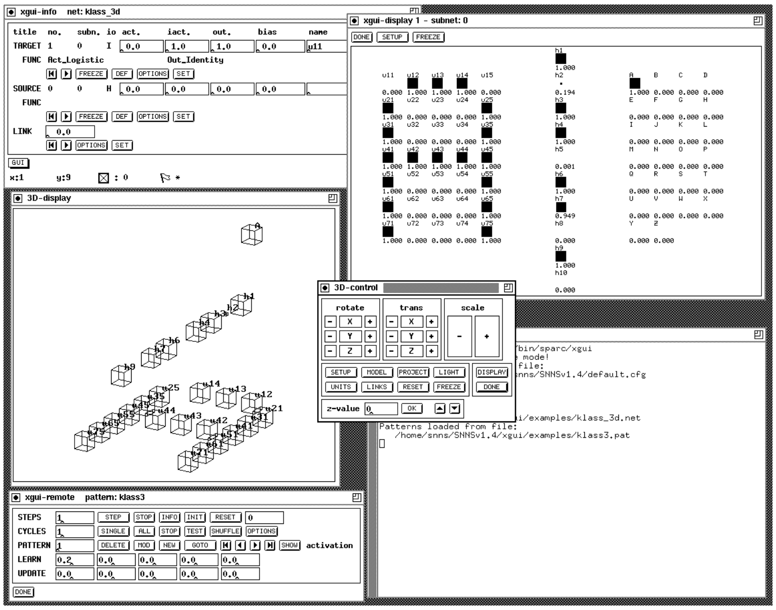

Figure 9.3: Network with display of activation

- the 2D → 3D transformation in the XGUI-display
- the 3D control panel
- the 3D display window

9.2.2 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.

9.2.3 Creating a 3D-Network

9.2.3.1 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 9.4 explains this behaviour.

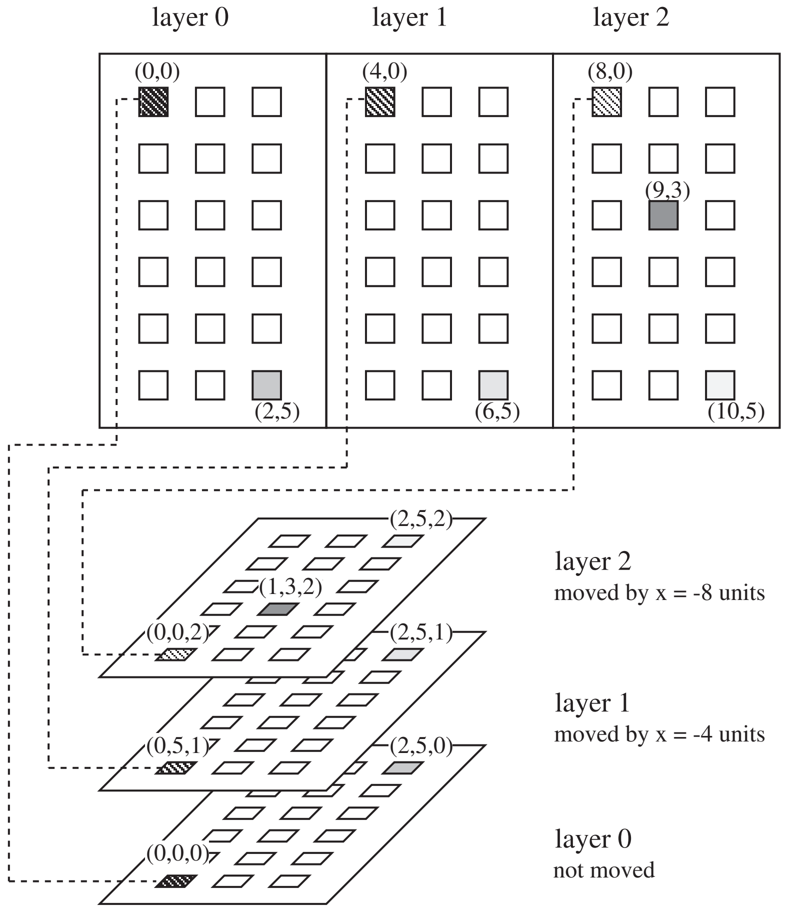

Figure 9.4: Layers in the 2D- and 3D-display

Therefore the network editor contains two new commands

Units 3d Z	: assigning a z-coordinate
Units 3d Move	: Moving a z-layer

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 9.2.3.4).

The user is advised to create a 3D network first as a wireframe model without links for much faster screen display.

9.2.3.2 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.

9.2.3.3 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.

9.2.3.4 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 `zvalue` 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 9.2.4.6)

9.2.3.5 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 9.5:

One scales the net with `scale` `-` in the transformation panel, then it looks like figure 9.6 (left). After a rotation with `rotate` `+` by 90° around the x-axis the network looks like figure 9.6 (right).

Now the middle layer is selected in the 2D-display (figure 9.7, left).

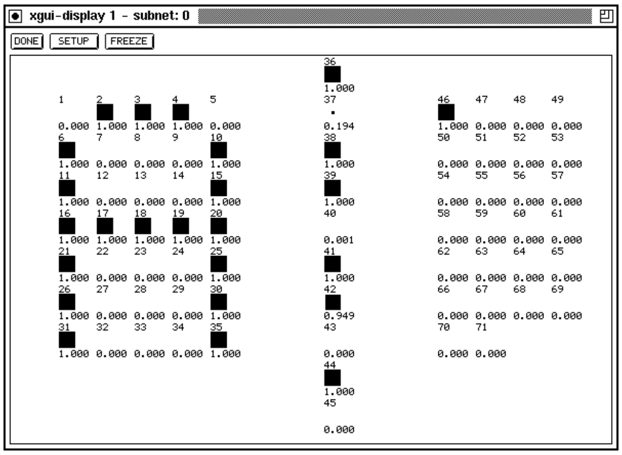

Figure 9.5: 2D-display

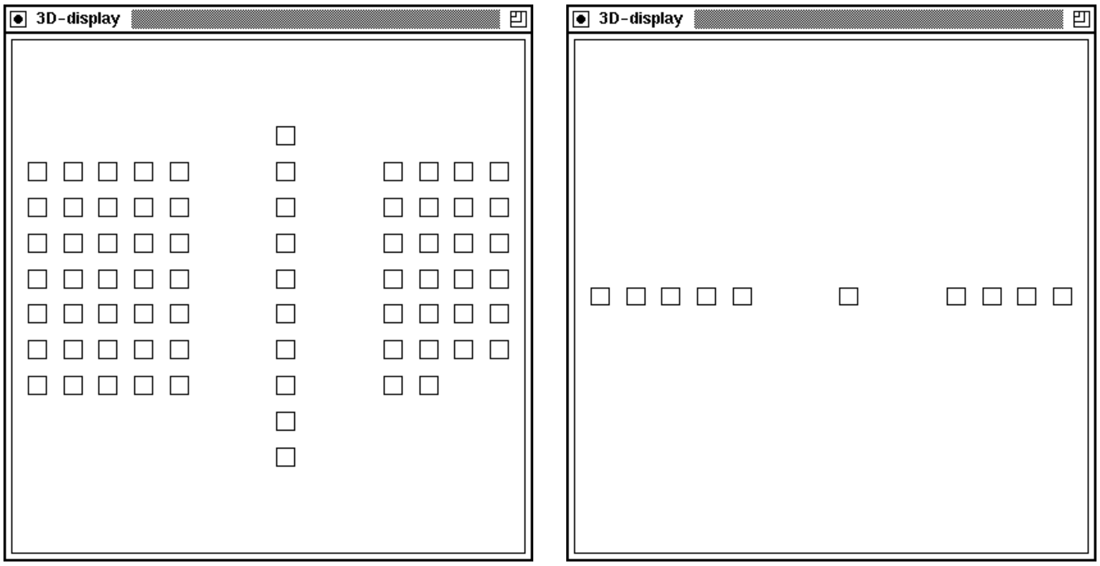

Figure 9.6: Scaled network (left) and network rotated by 90° (right)

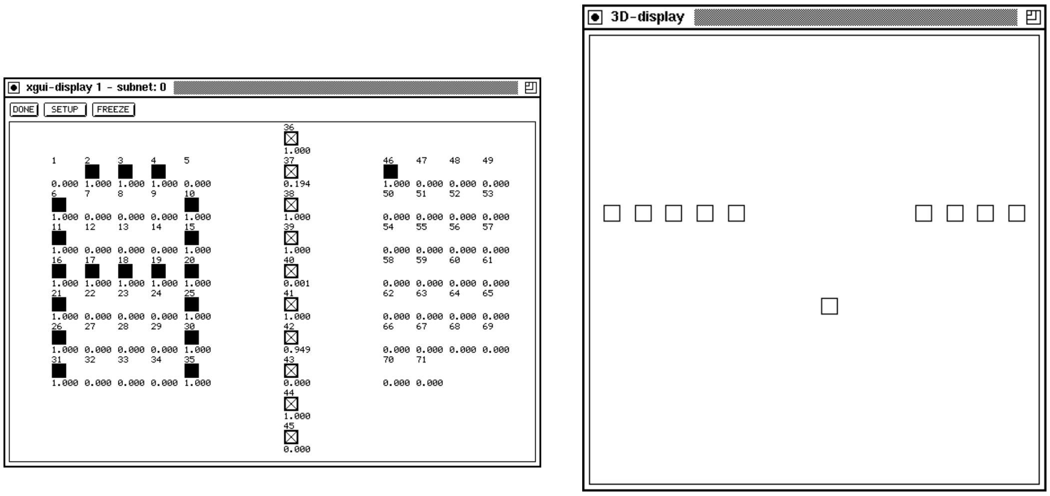

Figure 9.7: Selection of one layer (left) and assigning a z-value (right)

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 9.7 (right).

Now the reference unit must be selected (figure 9.8, 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 9.8 (right).

Figure 9.8: Selection of a reference unit (left) and moving a plane (right)

The output layer, which is assigned the z-value 6, is treated accordingly. Now the network may be rotated to any position (figure 9.9, left).

Finally the central projection and the illumination may be turned on (figure 9.9, right).

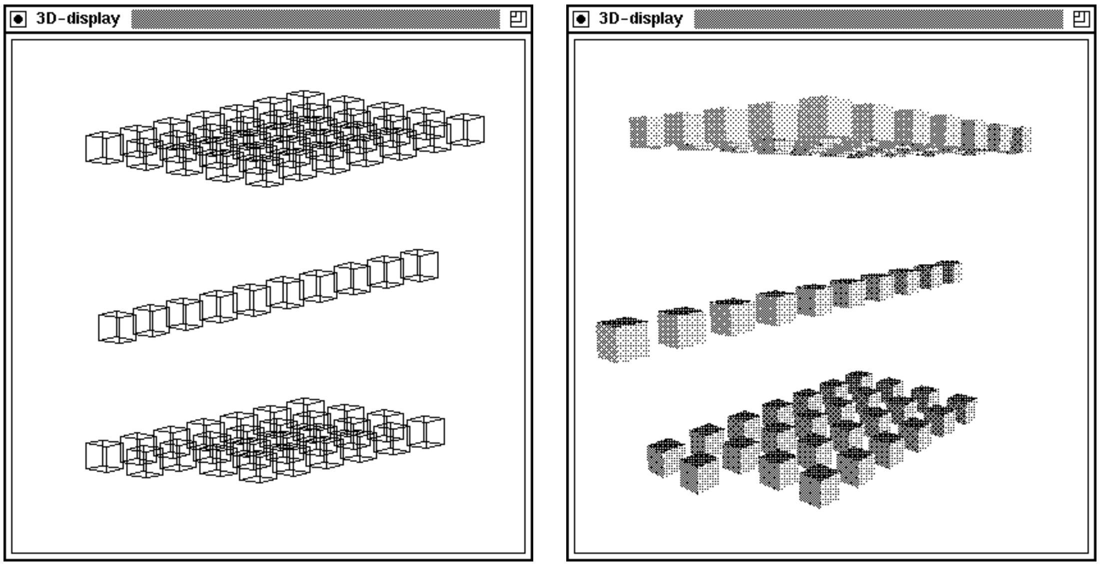

Figure 9.9: Wireframe model in parallel projection (left) and solid model in central projection (right)

These are the links in the wireframe model (figure 9.10, left).The network with links in the solid model looks like figure 9.10 (right).

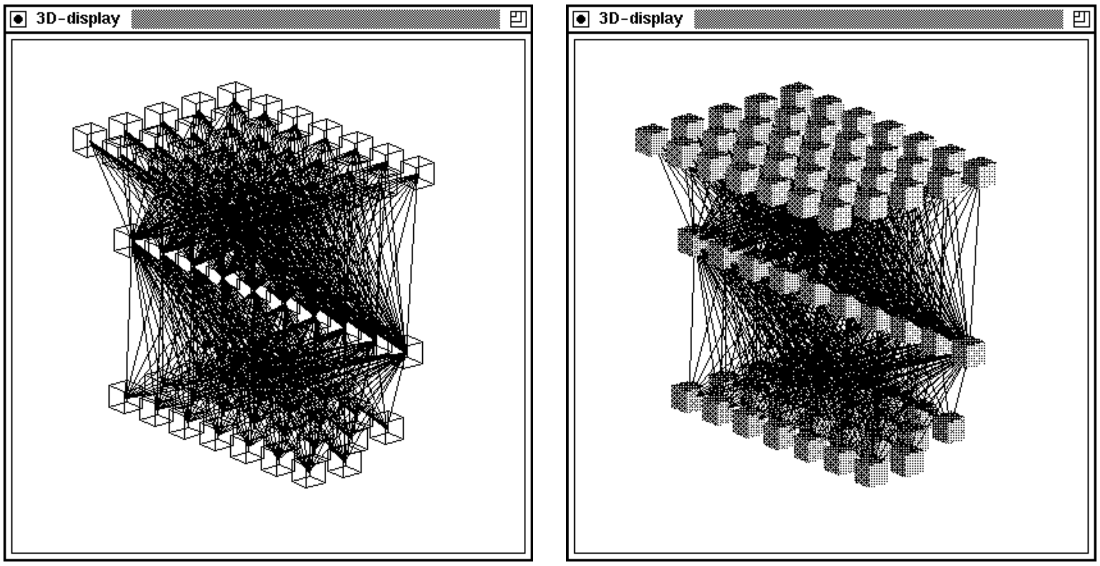

Figure 9.10: Network with links in the wireframe model (left) and in the solid model (right)

9.2.4 3D-Control Panel

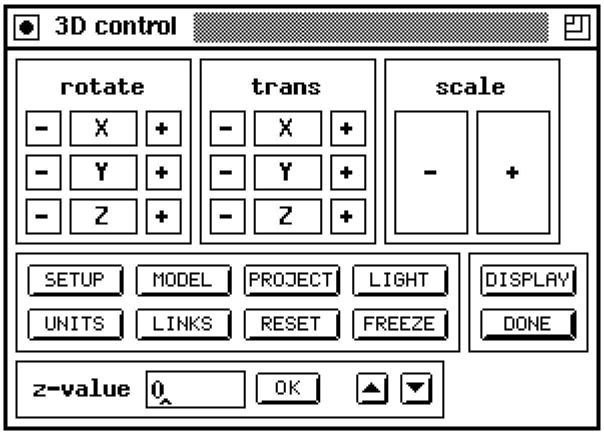

Figure 9.11: Control Panel

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

9.2.4.1 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 $\boxed{+}$ buttons rotate clockwise, the $\boxed{-}$ 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 $\boxed{+}$ buttons move to the right, the $\boxed{-}$ 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.

9.2.4.2 Setup Panel

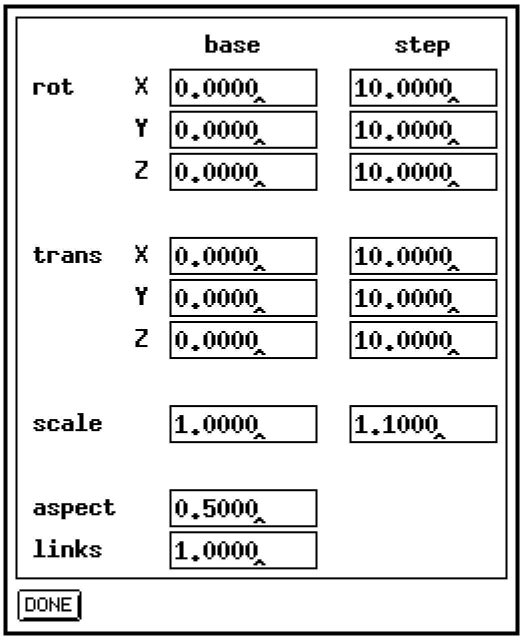

Figure 9.12: Setup Panel

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 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.

9.2.4.3 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 colour 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.

9.2.4.4 Project Panel

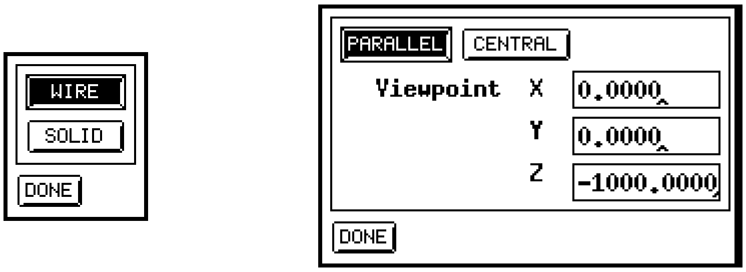

Figure 9.13: Model Panel (left) and Project Panel (right)

Here the kind of projection is selected.

PARALLEL selects parallel projection, i.e. parallels in the original space remain parallel.

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.

9.2.4.5 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.

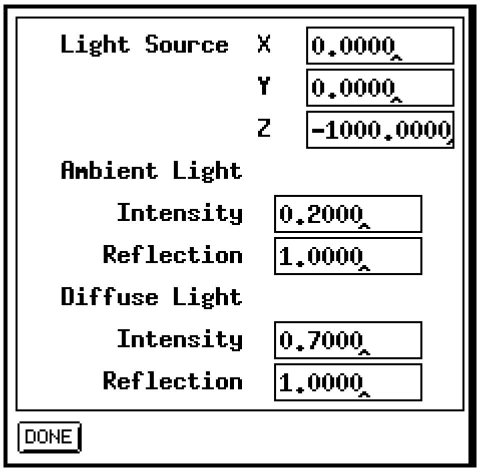

Figure 9.14: Light Panel

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)

9.2.4.6 Unit Panel

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

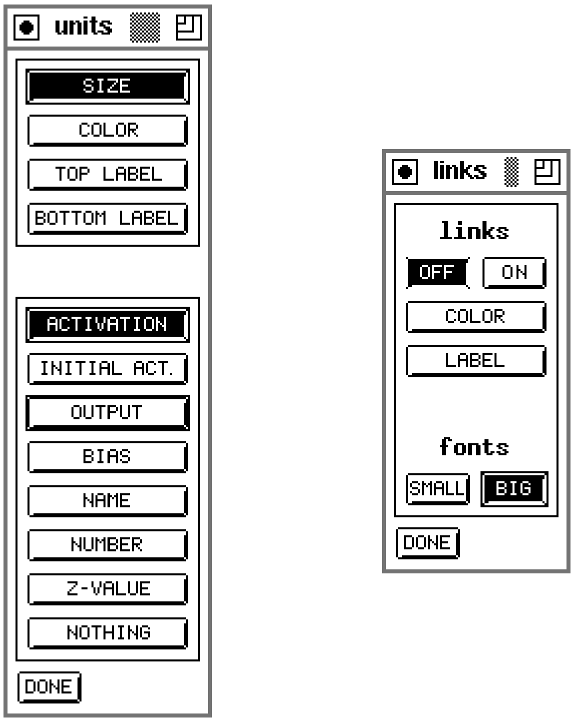

Figure 9.15: Unit Panel (left) and Link Panel (right)

- `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.

9.2.4.7 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

9.2.4.8 Reset Button

With the RESET button the values for moving and rotating are set to zero. The scaling factor is set to one.

9.2.4.9 Freeze Button

The FREEZE button keeps the network from being redrawn.

9.2.5 3D-Display Window

In the display window the network is shown. 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.

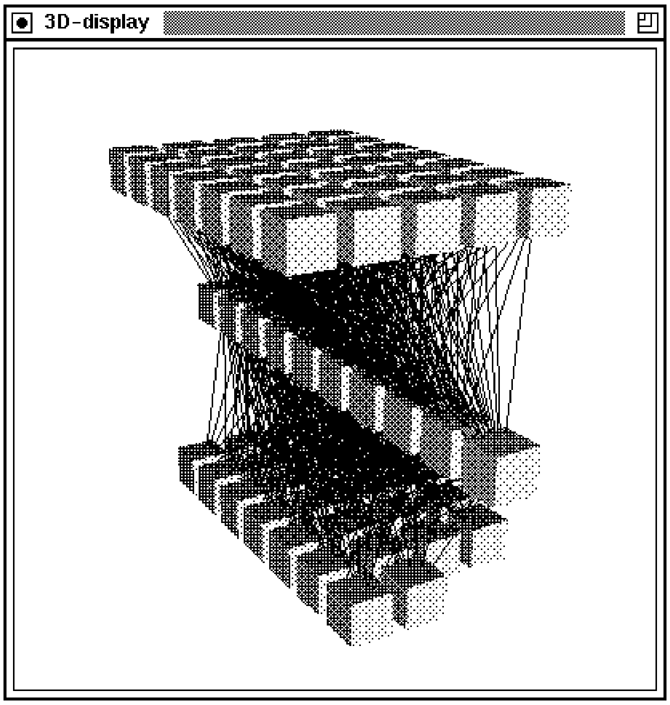

Figure 9.16: Display Window

Note that the 3D-display is only a display window, while the 2D-display windows have a graphical editor integrated.

Chapter 10

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.

10.1 The Snnsbat Environment

`snnsbat` runs very dependable even on instable 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'.

10.2 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> $\cdots$	'NoOfInitParam' parameters for the initialization function, separated by blanks.
LearnParam:	<float> $\cdots$	'NoOfLearnParam' parameters for the learning function, separated by blanks.
LearnPatternFile:	<string>	Filename of the learning patterns.
MaxErrorToStop:	<float>	Network error when learning can 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>	Number of parameters for the learning function.
PerformActions:	none	Execution run separator.
ResultFile:	<string>	Filename of the result file.
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.
ResultMinMaxPattern:	<int> <int>	Number of the first and last pattern to be used for result file generation.
Shuffle:	[YES | NO]	Flag for pattern shuffling.
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.

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 configuation 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 considerabe execution time. This allows for a continous 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:	Initialization function gets only zero values as parameters.
LearnParam:	Learning 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.
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.
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.
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, however, 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 appearence is taken.
# Keys that are not required for a special run may be omitted.
# If a key is ommited but required, a default value is assumed.
# The lines may be separated with comments.
#
# Please note the mandatory file type specification at the begining and
# the colon follwing the key.
#
#######################################################################
NetworkFile: /home/SNNSv3.0/examples/letters.net
#
InitFunction: Randomize_Weights
NoOfInitParam: 2
InitParam: -1.0 1.0
#
LearnPatternFile: /home/SNNSv3.0/examples/letters.pat
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, 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 V3.0 Batchlearning terminated.


SNNS V3.0 Batchlearning terminated at Tue Mar 23 08:49:04 1993
System:  SunOS Node:  matisse Machine:  sun4m
Networkfile './SNNS_BAAa02686' saved.
Logfile 'snnsbat.log' written.
```

10.3 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 '.../SNNSv3.0/kernel/bin/<machine_type>/'.
The sources of snnsbat can be found in the directory '.../SNNSv3.0/kernel/sources/'.
An example configuration file was placed in '.../SNNSv3.0/examples'.

Chapter 11

Design of the Simulator Kernel

11.1 Network Model of the Simulator

The network model, on which the simulator kernel is based, was designed to accommodate many network types. An important feature of the kernel is the complete encapsulation of all internal data structures and the efficient memory management.

The network model of the simulator kernel has the following properties:

- each unit has an arbitrary, user definable
  - *activation function* which takes the network input, the present activation, and the threshold to compute the new activation.
  - *output function* which computes the output value from the activation of the unit.
- each unit has an arbitrary number (possibly zero) of inputs which can be defined by the user. These so called *sites* have separate user definable input functions.
- sites may have an arbitrary number of connections from other units. A recurrent connection to the unit itself is possible.

11.2 Design Factors

The goal in designing the simulator kernel was to meet the following specifications:

- representation of a universal network model for small to medium sized networks (10^4 units, 10^6 connections)
- interactive manipulation of the net by the user
- encapsulation of all internal data structures with proper interfaces
- extendability

- very high efficiency
- modularity
- portability

The size of the networks that can be handled by the kernel is limited only by the size of (virtual) memory and the address space of the computer used. The simulator memory management relieves the operating system (UNIX), especially with larger nets. In interactive mode, the user has some powerful commands at hand to create and manipulate networks. These interface functions reduce the complex internal representation of the data to a representation at the logical network level.

Naturally, the demands of encapsulation and efficiency contradict each other. Nevertheless, a good compromise has been found: the functions, that can be defined by the user (activation and site functions)may be used with a macro library to access the kernel structures. This principle makes the combination of tight encapsulation and high execution speed possible. Measurements on several computer systems yielded the following propagation rates:

Machine type	**Operating System**	**Propagations** connections/sec. (CPS)	**Weight Updates** conn. updates/sec. (CUPS)
Intel 80486-33	Linux	453.000	194.000
DECserver 5400	Ultrix V4.2	398.000	175.000
DECstation 2100	Ultrix V4.2	494.000	211.000
DECstation 3100	Ultrix V4.2	689.000	287.000
DECstation 5000/200	Ultrix V4.2	733.000	248.000
SUN 3/60	SUN-OS 4.1.1	120.000	42.000
SUN SPARCstation ELC	SUN-OS 4.1.1	791.000	334.000
SUN SPARCstation 2	SUN-OS 4.1.1	950.000	415.000
SUN SPARCstation 10	SUN-OS 4.1.1	950.000	415.000
IBM RS 6000/320	AIX V.3.1	1.998.000	773.000
IBM RS 6000/320H	AIX V.3.1	2.207.000	814.000
HP 9000/720	HP-UX 8.07	1.707.000	764.000
HP 9000/730	HP-UX 8.07	2.147.000	1.093.000

The propagation rate measures the speed of the simulator in recall mode, i.e. the forward propagation rate. In this mode no weight updates take place. The usual measurement unit is connections/sec (CPS).

The weight update rate measures the speed of the training of a fully connected feedforward network trained with 'vanilla' (on line) backpropagation. Because there is a forward propagation, a backward propagation and a weight update phase for every pattern in each cycle, the weight update rates, measured in connection updates/sec. (CUPS) are usually lower by a factor of between 2 and 3 than the propagation rates.

These performance numbers have been obtained on machines running in our lab during normal use, with other users on the machines, with different main memory sizes and with the SNNS home directory mounted via NFS over the ethernet. Therefore, these numbers should only be regarded as performance indicators of SNNS but may not be quoted as machine architecture benchmarks.

The use of simulators for neural nets almost demands the use of parallel computers like the Connection Machine CM-2 [Hil85, HS86a, HS86b] or the MasPar MP-1, because of the inherent parallelism of the algorithm. The use of parallel computers will therefore surely increase the above values.

11.3 Layer Model of the Simulator Kernel

The simulator kernel is internally structured in three layers. Each higher layer represents a higher level of abstraction and is based on the layers below.

- The bottom layer is the SNNS memory management. It sits on top of the Unix memory management and offers functions to allocate and free data structures for the layer above.
- The next layer contains all functions to operate on, and to modify the network as well as propagation functions.
- The interface functions for the graphical user interface are located in the topmost layer.
- In the same layer as the function interface, there exists a file interface to the Nessus compiler.

In chapter 12 the data structures are described in detail. The description of the compiler interface can be found in appendix A.

Chapter 12

Internal Data Structures

The kernel needs numerous data structures to administer and display the network. The kernel has to display the units, links and sites (the *static* component), and to save the functions and connection weights of the units (the *dynamic* component). A symbol table is used to handle all the names and types associated with the units which are defined by the user. The allocation of data structures is done in large blocks of several hundred or even thousands of single structure components. This chapter describes all the relevant data types in detail.

12.1 Unit Array

The units and all their components are stored in the unit array. The data type 'array' was selected for storing the units, because of its short access time. When the array is filled up by requests from the user interface, the memory management automatically requests a new, bigger array from the operating system. The pointers to the structure components are reallocated, and the old empty array is returned to the operating system.

Each element of the unit array has the following components:

- *Out* stores the output value of the unit
- *flags* contains information for the memory management and the topological type of the unit (see chapter12.6)
- *lun* the logical unit number
- *lln* the logical number of the layer the unit belongs to
- **ftype_entry* points to the internal representation of the unit type, if a prototype exists for this unit
- *Aux* saves temporary values needed by internal kernel functions (e.g. backpropagation and topological sort)
- *TD* saves values needed for time delay networks

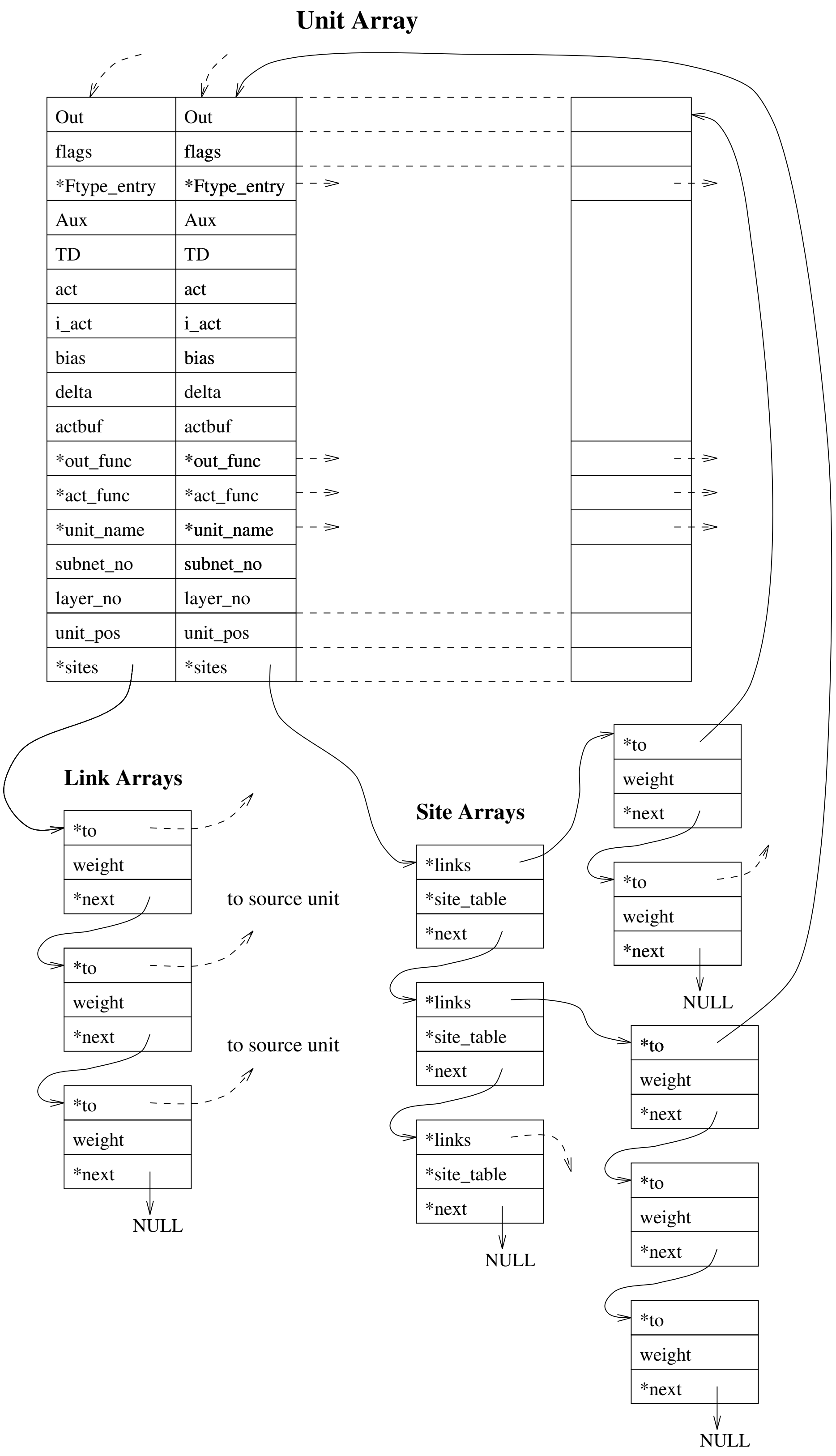

Figure 12.1: Internal network data structures (simplified)

- *act* contains the activation of the unit
- *i_act* saves the initial activation, used to reset the net
- *bias* contains the threshold of the unit
- *value_a, value_b, value_c* general purpose elements for the learning functions
- *olddelta, newdelta* error values in recurrent networks
- *actbuf* an array for storage of 10 miscellaneous values
- **out_func* points to the output function
- **act_func* points to the activation function
- **act_deriv_func* points to the derivation function of the activation function
- **unit_name* points to the unit name, a string in the symbol table
- *subnet_no* contains subnet membership information
- *layer_no* contains the location of the unit in a multilayer network
- *unit_pos* stores the position of the unit in the network
- **sites* points to a linked structure of input functions or connections

The internal representation of the network is displayed in figure 12.1.

12.2 Sites

The site data structure was designed to handle the input functions of the unit. The sites are constructed as a linear linked list. Several sites are grouped in an array (site array), each containing several hundred sites. The site arrays are also connected by a linked list. Thereby, empty, no longer needed, arrays can be freed.

The internal representation of a site has three components:

- **links* is a pointer to the input connections (links) of the site.
- **site_table* is a pointer to the so called *site table.* The site table contains information about the input function and the name of the site.
- **next* points to the next site of the unit. The list is terminated by NIL.

12.3 Links

The links make up the connections of the units. The direction of the internal pointers of the link data structure is from the target unit (destination unit) to the source unit (origin unit). Note that this is reversed from the direction of the links in the neural network (which run from source to target unit). This approach allows the efficient storage of the

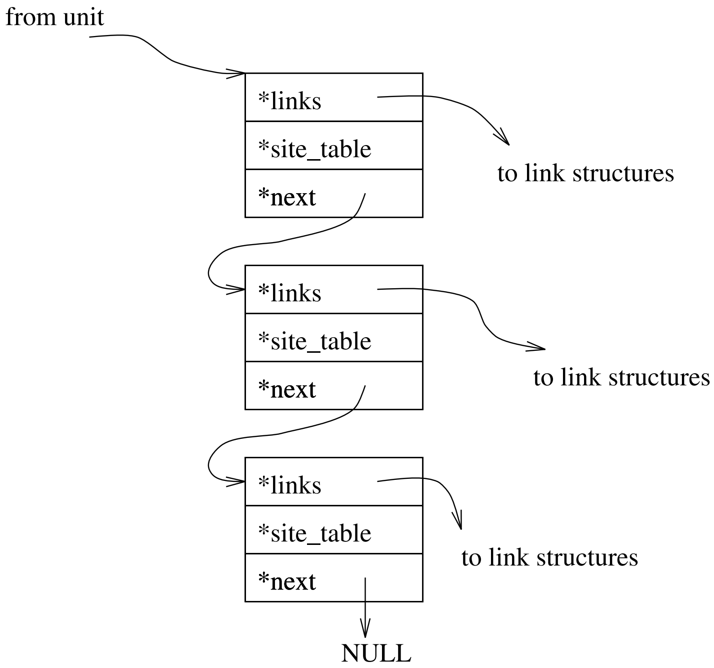

Figure 12.2: Site list

network structure with a minimum of memory. However, the memory needed to store big nets is still the limiting factor. For a net with 10.000 units with a connectivity of 20%, 20 million connections have to be stored, requiring 230 megabyte of memory. In comparison, the fraction of the memory used to store all other components is negligible.

The structure of a link is quite similar to that of a site:

- **to* points to the source unit of the link.
- *weight* contains the connection weight.
- *value_a, value_b, value_c* general purpose elements for the learning functions
- **next* points to the next connection of the same target unit (terminated by NIL).

Like sites, the links are grouped in linked arrays which are handled by the SNNS memory management routines.

12.4 Network Memory Management

When a network is loaded or constructed interactively, the memory management always has to supply enough units. As already mentioned, the user doesn't have to care about memory allocation and deallocation. During the interactive manipulation of the net, the user can change or delete whole parts of the net. The empty components are grouped in a linked free storage list for later reuse. This is also true for tables like the symbol table. There, identical names (identifiers) are stored only once. A reference counter decides

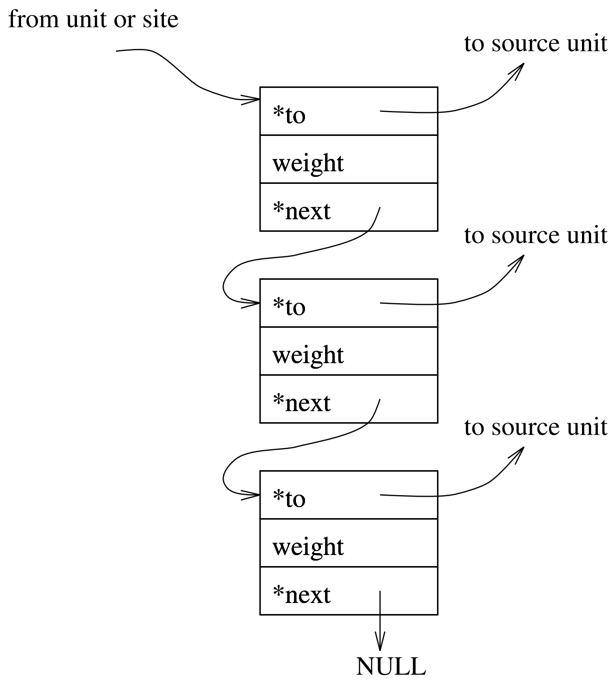

Figure 12.3: Link list

whether an entry can be freed on a delete operation or not. An exhaustive garbage collection is performed when the whole net is deleted.

12.4.1 Link/Site Arrays

The memory management is presented in figure 12.4 with the example of link arrays. Each first entry is used for backward links. The data structure of site arrays is similar.

12.4.2 Symbol Table

The symbol table administers the identifiers of the units, sites and unit prototypes. An entry in the table is deleted when the reference counter is zero. Figure 12.5 shows the symbol table with linked empty entries.

12.5 Unit Flags

The simulator computes the topological position (topological type) and the current state of the unit with the help of the unit flags. The topological type is important for the teaching and propagation algorithms. With certain status bits, the state of the unit in the network is checked. The flags are stored as bit patterns in the unit array. Figure 12.6 shows the correspondence between bits and flags of the unit entry.

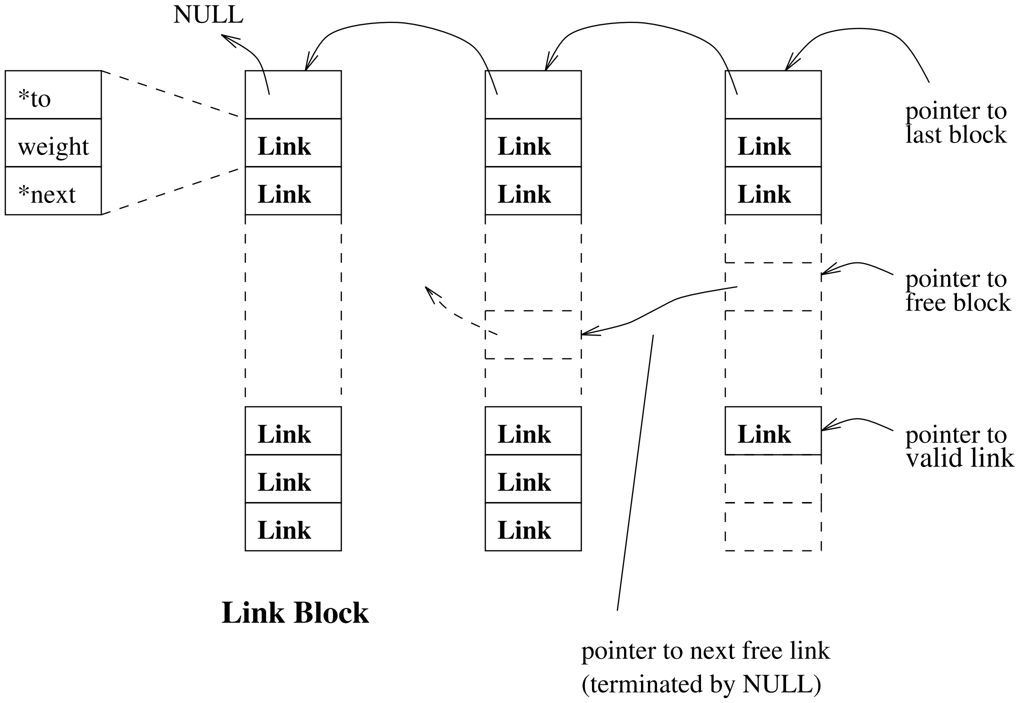

Figure 12.4: Link arrays

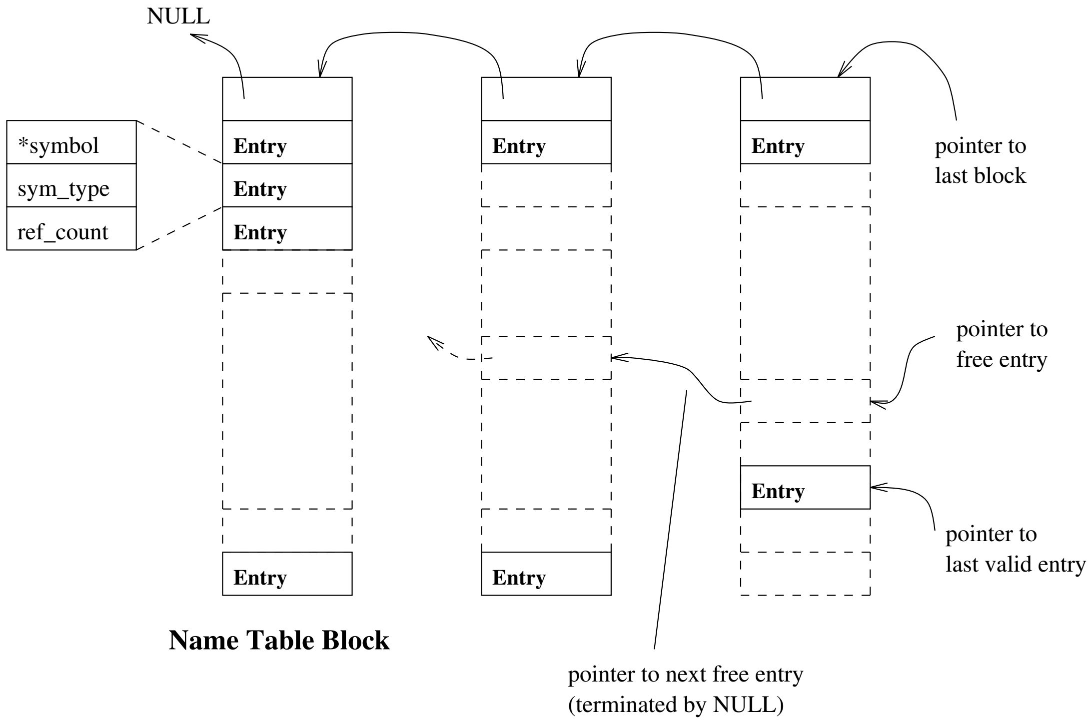

Figure 12.5: Symbol table

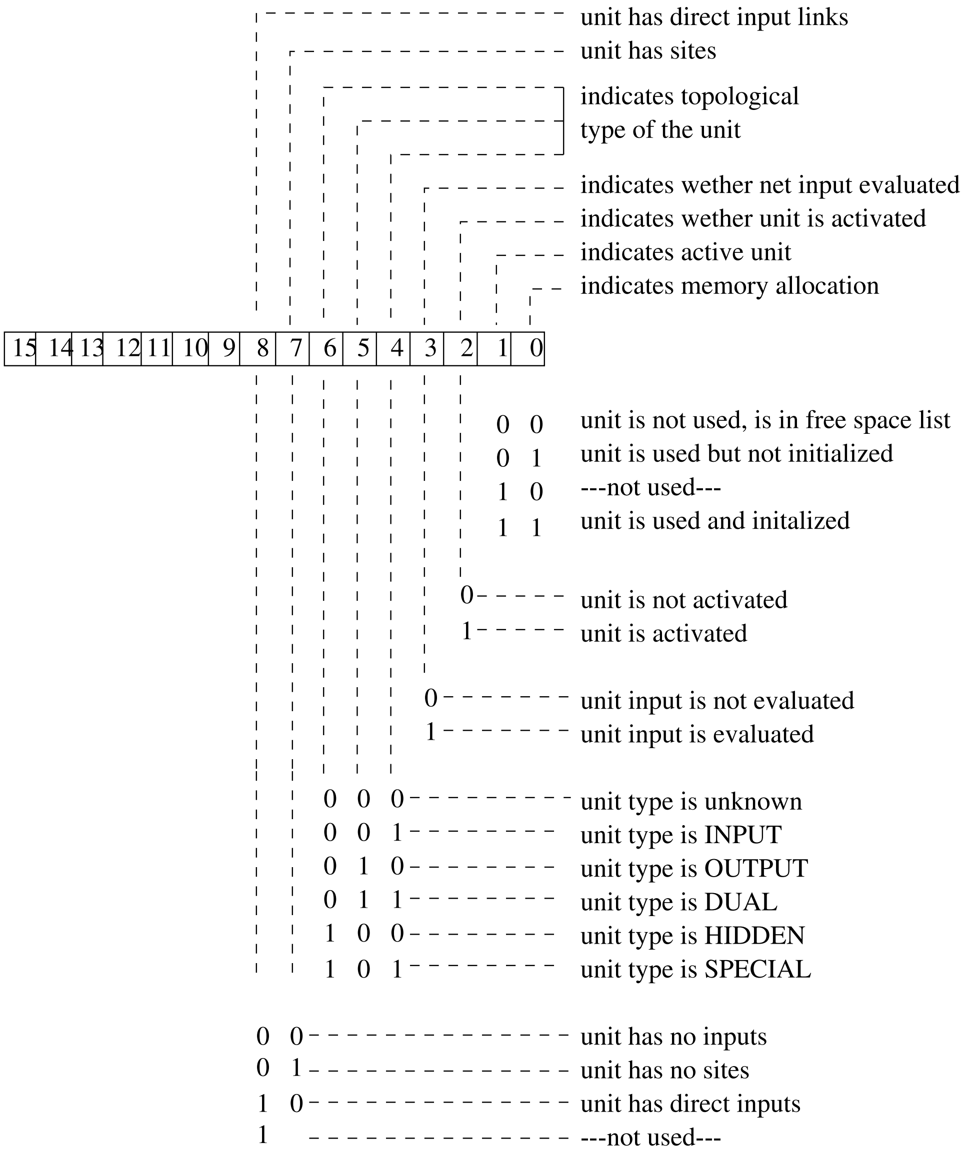

Figure 12.6: Unit flags

12.6 Function Table

The various site and unit functions are stored in the function table. The user can define his own functions in form of a C program. Information about these functions (name, type) is stored in a table, to give the kernel access to and use of these functions.

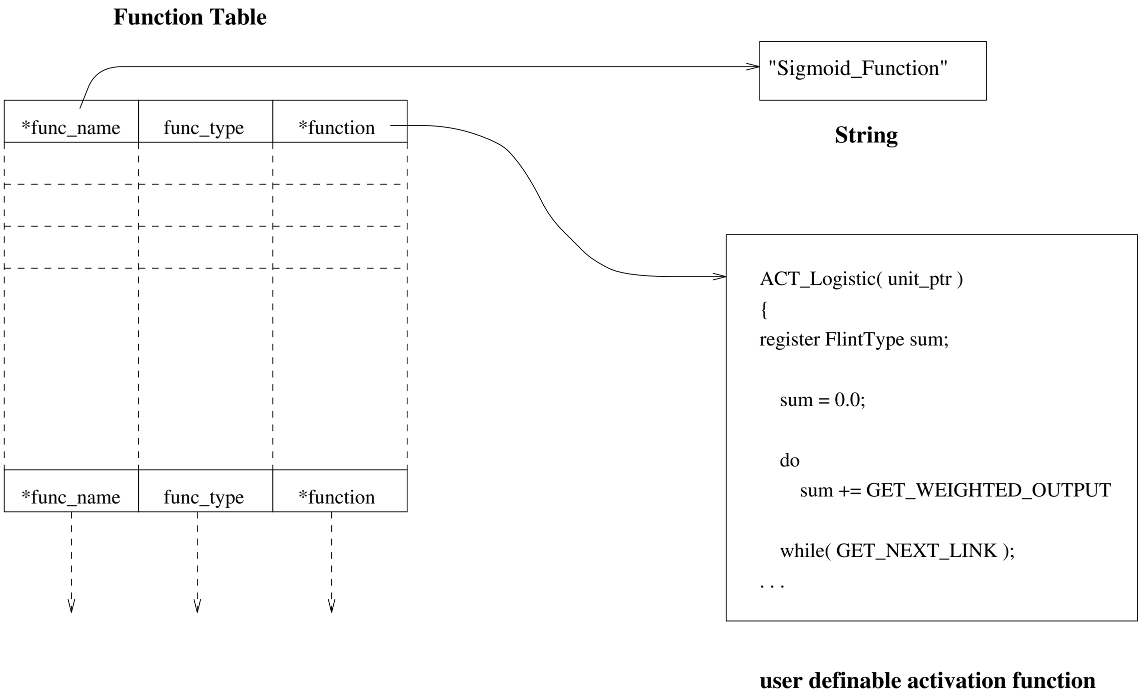

Figure 12.7: Function table

Chapter 13

Kernel Function Interface

13.1 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).

13.2 Unit Functions

The following functions are available for manipulation of the cells and their components:

```
krui_getNoOfUnits()
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_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[] )
```

13.2.1 Unit Enquiry and Manipulation Functions

`int  krui_getNoOfUnits()`
determines the number of 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.

`krui_err  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; returns an error code if no units are defined.

`krui_err  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 )
void       krui_setUnitActivation( int UnitNo, FlintType 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 )
void  krui_setUnitOutput( int  UnitNo, FlintType  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,
                                     unsigned short layer_bitField )
```
returns/sets the layer number (16 Bit integer).

```
void  krui_getUnitPosition( int  UnitNo, struct PosType  *position )
void  krui_setUnitPosition( int  UnitNo, struct PosType  *position )
```
returns/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( struct TransTable * *xy_trans_tbl_ptr )
```
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.

```
void  krui_freezeUnit( int  UnitNo )
```
freezes the output and the activation value of the unit, i.e. these values are not updated anymore.

```
void  krui_unfreezeUnit( int  UnitNo )
```
switches the computation of output and activation values on again.

```
bool  krui_isUnitFrozen( int  UnitNo )
```
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`.

13.2.2 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

[1]The term T-type was changed to IO-type after completion of the kernel

- 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, FlintType  act,
                      FlintType  i_act, FlintType  out,
                      FlintType  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.

13.3 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()
```

13.3.1 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.

13.3.2 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() ) { }
```

13.4 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( FlintType  strength )
krui_createLink( int source_unit_no, FlintType  strength )
krui_deleteLink()
krui_deleteAllInputLinks()
krui_deleteAllOutputLinks()
krui_jogWeights( FlintType  min, FlintType  max)
```

`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( FlintType  strength )
```
determines/sets the connection weight of the current link.

`krui_err krui_createLink( int source_unit_no, FlintType strength )`
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.

`void 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.

```
void  krui_deleteAllInputLinks()
void  krui_deleteAllOutputLinks()
```
deletes all inputs/outputs at the current unit/site.

`void krui_jogWeights( FlintType min, FlintType max)`
adds values, randomly distributed in $[min, max]$, to the connection weights of the network. See also `krui_setSeedNo(...)`.

13.5 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.

13.6 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.

13.7 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.

13.8 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_testNet( int  pattern_no,
                       float *updateParameterArray, int NoOfUpdateParams,
                       float *parameterInArray, int  NoOfInParams,
                       float * *parameterOutArray, int  *NoOfOutParams )
```
calculates the network error with the given pattern. Uses the current update function to propagate the network. `UpdateParameterArray` contains the parameters of the update function. `NoOfUpdateParams` contains the number of input parameters of the update function. `parameterInArray[0]` contains the max. deviation. Set `NoOfInParams` to 1. `parameterOutArray[0]` returns the error of the network. `parameterOutArray[1]` returns the number of output units with a higher error value than the given max. deviation. `NoOfOutParams` is set to 2.

Note: Patterns must be loaded before calling this function. The functions returns an error code if an error occurred.

`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).

13.9 Learning Functions

`krui_err krui_setLearnFunc( char *learning_function)`
selects the learning function; returns an error code if the given learning function is unknown.

`char *krui_getLearnFunc( void )`
returns the name of the current learning function. The default learning function is *'Std_Backpropagation'* (see also `kr_def.h`).

```
krui_err  krui_learnAllPatterns( float *parameterInArray,
                                 int  NoOfInParams,
                                 float  * *parameterOutArray,
                                 int *NoOfOutParams )
```
learns all patterns (each consisting of an input/output pair) using the current 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 )
```

same as `krui_learnAllPatterns( ... )` but teaches only the current pattern.

13.10 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_shufflePattern( 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(...)`.

`int  krui_getNoOfPatterns( void )`
returns the actual number of patterns (0 if no patterns have been loaded).

13.11 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_loadPatterns( char *  filename)
krui_err  krui_savePatterns( char *  filename)
```
loads/saves the patterns from/in memory.

An error code is returned if an error occurred during the execution of the I/O functions.

13.12 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.

13.13 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( FlintType  act, FlintType  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

```
void  krui_resetNet()
```
sets the activation values of all units to their respective defaults.

13.14 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.

13.15 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_getClassificationStatus( 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.

`int artui_getN ( void )`
returns the number of existing F_1 units (N) as function value. The function can be used for ART1 and ART2 models, **not** for ARTMAP models.

`int artui_getM ( void )`
returns the number of existing F_2 units (M) as function value. The function can be used for ART1 and ART2 models, **not** for ARTMAP models.

`int artui_getNa ( void )`
returns the number of existing F_1^a units (N^a) of the ART^a subnet of an ARTMAP network as function value. The function can **only** be used for ARTMAP models.

`int artui_getMa ( void )`
returns the number of existing F_2^a units (M^a) of the ART^a subnet of an ARTMAP network as function value. The function can **only** be used for ARTMAP models.

`int artui_getNb ( void )`
returns the number of existing F_1^b units (N^b) of the ART^b subnet of an ARTMAP network as function value. The function can **only** be used for ARTMAP models.

`int artui_getMb ( void )`
returns the number of existing F_2^b units (M^b) of the ART^b subnet of an ARTMAP network as function value. This is the number of MAP field units, too. The function can **only** be used for ARTMAP models.

13.16 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 not update unit because unit is frozen
KRERR_REDEF_SITE_NAME:   Redefinition of site name is 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:   This 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:  In-Core 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_PERC_SITES:       Learning function does not support sites
KRERR_NO_OF_UNITS_IN_LAYER: Wrong no. of units in layer:
KRERR_UNIT_MISSING:     Unit is missing or not correctly connected:
KRERR_UNDETERMINED_UNIT: Unit doesn't 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 destination unit:
KRERR_TOPOLOGY:         This network is not fitting the required topology
KRERR_PARAM_BETA:       Wrong beta parameter in unit bias value:
```

Chapter 14

Transfer Functions

14.1 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.

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	$a_j(t) = net_j(t) * \theta$
Elliott	$a_j(t) = \frac{net_j(t)}{1+\lvert net_j(t)\rvert}$
Identity	$a_j(t) = net_j(t)$
IdentityPlusBias	$a_j(t) = net_j(t) + \theta$
Logistic	$a_j(t) = \frac{1}{1+e^{-net_j(t)+\theta}}$
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	$a_j(t) = \begin{cases} 1 & \text{if } net_j(t) \geq \theta \\ 0 & \text{if } net_j(t) < \theta \end{cases}$
Product	$a_j(t) = \prod_i w_{ij} o_i$
RBF_Gaussian	see chapter 8.10.2.1
RBF_MultiQuadratic	see chapter 8.10.2.1
RBF_ThinPlateSpline	see chapter 8.10.2.1
Signum	$a_j(t) = \begin{cases} 1 & \text{if } net_j(t) > 0 \\ -1 & \text{if } net_j(t) \leq 0 \end{cases}$
Signum0	$a_j(t) = \begin{cases} 1 & \text{if } net_j(t) > 0 \\ 0 & \text{if } net_j(t) = 0 \\ -1 & \text{if } net_j(t) \leq 0 \end{cases}$
StepFunc	$a_j(t) = \begin{cases} 1 & \text{if } net_j(t) > 0 \\ 0 & \text{if } net_j(t) \leq 0 \end{cases}$
TanH	$a_j(t) = \tanh(net_j(t))$
TanH_Xdiv2	$a_j(t) = \tanh(net_j(t)/2)$

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_NormP`, `ART2_NormV`, `ART2_NormW`, `ART2_NormIP`, `ART2_Rec`, `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`	$o_j(t) = \begin{cases} 0 & \text{if } a_j(t) \leq 0 \\ a_j(t) & \text{if } a_j(t) < 1 \\ 1 & \text{falls } a_j(t) \geq 1 \end{cases}$
`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.

14.2 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 "`.../SNNSv3.0/kernel/func_tbl`" the new function has to be defined as:
   extern FlintType My_fancy_function();
2. In the same file 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")!
3. The functions must be implemented as C programs in following files:

activation functions in "`.../SNNSv3.0/kernel/trans_f.c`"
output functions in "`.../SNNSv3.0/kernel/trans_f.c`"
site functions in "`.../SNNSv3.0/kernel/trans_f.c`"
initialization functions in "`.../SNNSv3.0/kernel/init_f.c`"
learning functions in "`.../SNNSv3.0/kernel/learn_f.c`"
update functions in "`.../SNNSv3.0/kernel/update_f.c`"

The name of the implemented function has to match the name specified in the function table!

4. Recompile the kernel with make. Make sure, that "`ar libfuncs.a`" and "`ranlib libfuncs.a`" is executed.

5. Recompile and link the user interface using make.

The new function should be available now in the user interface together with all predefined functions.

Chapter 15

Simulator Kernel Implementation

The simulator kernel alone, including comments, has a volume of about 60000 lines of source code with about 1.650 Mbyte. It is implemented in ANSI C. The simulator kernel consists of the following files:

Header files:

All source code files have corresponding public (.h) and private (.ph) header files associated with them. The public header files are used to make the exported functions of the modul accessible to the other moduls, while the private ones define function prototypes and variables exclusively global to the functions of the modul. Additionally there are the following header files without corresponding c-moduls.

Source file	**Meaning**
art_typ.h	global types for all ART moduls
cc_mac.h	macros for cc and rcc moduls
cc_typ.h	global types for all cc moduls
func_mac.h	macros for transfer functions
glob_typ.h	user interface data types
kr_const.h	constants of the kernel
kr_def.h	defaults for the kernel
kr_mac.h	macros of the kernel
kr_typ.h	data types of the kernel
krart_df.h	macros for the ART kernel functions
krui_typ.h	user interface functions
random.h	randomize functions for System V
version.h	version and patchlevel of the kernel and the kernel interfaces

Sourcecode files:

Source file	**Meaning**
art_ui.c	ART user interface functions
arttr_f.c	special transfer functions for ART models
cc_learn.c	learning functions for cascade correlation
cc_rcc.c	Common functions of CC and RCC
cc_rcc_topo.c	CC and RCC topology check
func_tbl.c	contains the function table
init_f.c	user definable initialisation functions
kernel.c	kernel low-level functions
kr_amap.c	kernel low-level functions for ARTMAP model
kr_art.c	kernel low-level functions for ART models
kr_art1.c	kernel low-level functions for ART1 model
kr_art2.c	kernel low-level functions for ART2 models
kr_funcs.c	routines to handle the function table
kr_inversion.c	network inversion functions
kr_io.c	compiler-kernel interface (file I/O)
kr_mem.c	functions of the memory management
kr_rand.c	random functions for MS-DOS implementation
kr_td.c	time delay learning functions
kr_ui.c	high level user interface functions
learn_f.c	user definable teaching functions
make_tbl.c	create a lookup table for transfer functions
matrix.c	matrix handling functions
rcc_learn.c	learning functions for recurrent cascade correlation
strdup.c	contains the strdup(...) function (not in ULTRIX-32)
tbl_func.c	sigmoid activation function with table lookup
trans_f.c	user definable transfer functions (unit, site, and output functions)
update_f.c	user definable update functions

Test, demo, and benchmark programs:

Source file	**Meaning**
bignet.c	demo program to generate large 3-layer feedforward networks
m_art.c	network generator for ART network architectures
netlearn.c	demo program for the network training
netperf.c	SNNS kernel benchmark test program
snnsbat.c	SNNS kernel network training program for very large networks and/or training sets. Runs in the background and controls training of the network (See also dokument file: snnsbat.doc).

Example Networks:

Source file	**Meaning**
art1.net	ART-1 network
art2.net	ART-2 network
artmap.net	ARTMAP network
encoder.net	8-3-8 encoder network
font.net	16x16 character recognition network
letters.net	character recognition network
letters3D.net	character recognition network in 3D
necker.net	character comparison network
nettalk.net	nettalk network
xor.net	XOR Network
xor-rec	recurrent XOR Network

Probably there will be more example networks in the new SNNS release. See the files in the examples directory.

Chapter 16

Implementation of the User Interface

This chapter explaines important data types, functions, and the program structure of the graphical user interface. Definitions and data structures used by the program, are located not only in the file `glob_typ.h` of the SNNS kernel, but also in the header files of the X-window system and the Athena Toolkit. For reference of those definitions see the X11 documentation release 4.0 or 5.0.

How X, XGUI and the kernel cooperate is sketched in figure 16.1. The MIT Athena Toolkit was selected because of its widespread use. In this toolkit, so called *widgets* are defined (window elements like boxes, menu buttons, toggles, scrollbars and so forth). XGUI should work with all window mangers, but the following descriptions refer only to the *twm* window manager.

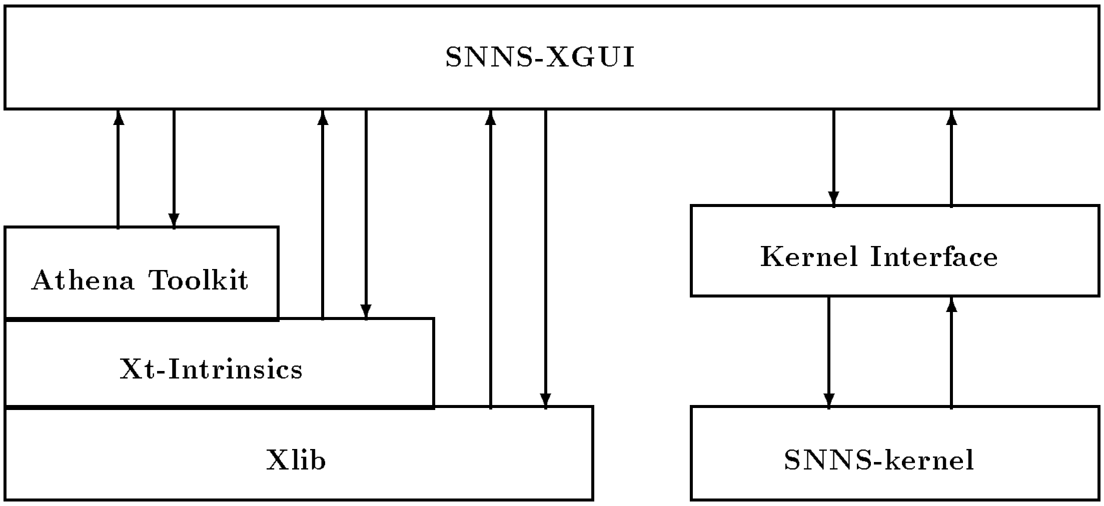

Figure 16.1: How X, XGUI and the simulator kernel cooperate

After opening of the first window, SNNS-XGUI transfers control to X by entering the main-event-dispatch-loop[1].To react to different events (e.g. mouse movement), X now calls predefined subroutines like callbacks and event handlers, which return control to X after completion. All outputs (e.g. line drawing) are buffered by the X server until it has resumed control. This asynchronous behaviour complicates debugging, since changes in the user interface are invisible at the time they are processed. The compile switch `DEBUG` forces the X server with calls to `XFLUSH()` to produce output at crucial points. The same result is obtained by specifying `-synchronous` as an argument in the call to XGUI. In both cases output speed decreases dramatically.

For windows which return a value, like the Layer window or the Confirmer, X must react directly to events. There, the Event-Dispatch-Loop is re-programmed where the exit condition is true after the call to a routine for a specific button (e.g. `DONE`). The next program segment can immediately use the result. The call to the confirmer demonstrates this very clearly:

```
if (ui_confirmYes(''Load will erase current network. Load?''))
    ui_file_loadNet(filename);
else
    ui_printMessage(''Loading aborted.'');
...
```

`ui_confirmYes()` builds the confirmer and dispatches all events until one of the buttons `YES` or `NO` is pressed. Then it returns this value as a boolean. Without this loop, the confirmer would become visible only after the processing of the code, and a default value would always be returned!

On processing the teaching cycles or the update steps, it is necessary to check for new events after each step. This is the only way to recognize the clicking of the `STOP` button in time. The *WorkProc* construct is a means for solving this problem. The routine for performing one step is declared as WorkProc with the statement `XtAppAddWorkProc(...)`. This routine is called as long as no events are waiting. When all cycles/steps are processed it returns `True`, thereby being removed. A soon as the button `STOP` is pressed, the corresponding callback routine `ui_rem_stopProc` removes the WorkProc with the command `XtRemoveWorkProc(ui_workProcId)`. A WorkProc, already existing when a new one is defined, gets deleted (see also [You89], p. 136ff). This also explains, why the starting of update steps cancels a running teaching process and vice versa. A running WorkProc always terminates correctly, i.e. it is not interrupted by incomming events. Therefore XGUI processes all events as usual, but slows down with increasing running time of the WorkProcs.

In general there is one callback function for each button. However, some buttons call the same function, and use just different values for the argument `client_data`. This argument, the calling widget and an aditional argument `call_data` are passed over to the callback procedure. `client_data` is defined by `AtAddCallback()`, `call_data` by the kind of widget that is calling. The scrollbar widget, for example, passes the position of the slider.

[1]SNNS-XGUI interpretes this as an application.

With the use of window systems like X, a structuring of the source code according to the windows used is almost automatic (see also table 16.1 and 16.2).

Callback procedures are kept in a separate file. Since SNNS-XGUI consists mainly of those functions and calling depth remains fairly shallow, the source code is easily readable.

16.1 Administration of the Windows

There are three kinds of windows. They differ in the shell selected, and in the mode of the event handling. Transient shell windows (`transientShellWidgetClass`) have only a thin frame and block all other windows. This means that no more events are processed until this window is closed and the shell widget is destroyed. This is done by generating the window with the call `Xt_popup(widget, XtGrabExklusive)` with the toplevel widget `ui_toplevel` as the parent. Some windows, like the file panel, do not block XGUI, because they are called with `Xt_popup(widget, XtGrabNone)`. The third kind of windows are created as a top level shell (`topLevelShellWidgetClass`) like the manager panel.

Most of the windows (AsciiTextWidget, ToggleWidget, ScrollbarWidget) have panel items that can be changed by the user. These values are commonly stored in global variables, to enable other parts of the program to access them, even after the widget is destoyed. Some resources are predefined as so called *fallback resources* in the file `ui_main.h`. The bitmaps for the buttons are located in the directory `iconsXgui` and are linked to the program at compile time.

A callback routine is called every time a window is destroyed (for example by clicking `DONE` in popups). This routine takes care of all necessary actions, before vital information is lost, like copying the values of AsciiTextWidgets into global variables. In addition, there are flags which indicate whether or not a popup is existing.

Many panels contain form widgets, where other widgets can be positioned freely. In XGUI the location of these new widgets is always relative to the position of already existing widgets. The info panel, for example, takes the first line as horizontal reference. The change of widget positions has to be made very carefully, since some widgets have the same variable[2] and may not be addressable anymore.

To ease the handling of the widgets some routines are given in the file `ui_xWidgets.c`. They allow the creation of Label-, AsciiText-, Button-, MenuButton-, Toggle- and Scrollbar-Widgets in a form widget. Three functions return the value of an ascii text widget as a string (`ui_xStringFromAsciiWidget`), float (`ui_xFloatFromAsciiWidget`) or integer (`ui_xIntFromAsciiWidget`). To set the contents of a widget to a specific value use `ui_xSetString`, or `ui_xSetLabel`. To set/read the state of a toggle use `ui_xSetToggleState` and `ui_xGetToggleState`.

[2] Otherways every widget would need its own local variables

bn_TD_bignet.c	the BIGNET time delay panel & functions
bn_art1.c	the BIGNET ART1 panel & functions
bn_art2.c	the BIGNET ART2 panel & functions
bn_artmap.c	the BIGNET ARTMAP panel & functions
bn_basics.c	subroutines common to all BIGNET tools
bn_bignet.c	the BIGNET feed forward panel & functions
bn_menu.c	the BIGNET tools selection menu
cc_main.c	cascade correlation control panel
d3_anageo.c	3–D matrix operations
d3_disp.c	event loop for 3–D display
d3_dither.c	color rastering
d3_draw.c	high level drawing functions
d3_font5x7.c	5x7 font array
d3_font5x8.c	5x8 font array
d3_font8x14.c	8x14 font array
d3_fonts.c	text output control
d3_global.c	global variables for 3–D
d3_graph.c	low level drawing functions
d3_light.c	illumination panel
d3_links.c	3–D link display panel
d3_lists.c	3–D list managenemt
d3_main.c	3–D interface handling
d3_model.c	solid/wire display setting
d3_move.c	rotation,translation and scaling
d3_pannels.c	3–D control panel
d3_point.c	3–D pixel output
d3_project.c	3–D projection panel
d3_setup.c	3–D setup panel
d3_shade.c	shading algorithm
d3_units.c	3–D unit setup panel
d3_xUtils.c	X window routine interface
d3_zValue.c	z–value input panel
d3_zgraph.c	z–buffer functions
o_graph.c	plotting of the error graph
ui_action.c	editor actions
ui_colEdit.c	changing of the display colors
ui_color.c	handles the color maps

Table 16.1: The source files of SNNS-XGUI, Part I

ui_config.c	load & save of configurations
ui_confirmer.c	handling of the confirmer
ui_display.c	administration of displays
ui_displwght.c	implements the Hinton– and WV–diagrams
ui_edit.c	editing of f-types and sites
ui_event.c	event handler for mouse or window events
ui_file.c	creation of the file panel
ui_fileP.c	load, save (high level)
ui_funcdispl.c	displays the activation and output functions
ui_info.c	creation of the info panel
ui_infoP.c	callbacks and handling of the info panel
ui_inversion.c	handels the inversion display & algorithm
ui_key.c	event handler for keyboard
ui_layer.c	creation of layer popups
ui_layerP.c	handling of the layer popup
ui_lists.c	administration of the list popup
ui_main.c	initializing
ui_mainP.c	windows and popups, help facility
ui_maspar.c	maspar user interface
ui_netGraph.c	drawing of network elements (mid level)
ui_netUpdate.c	drawing of the network, whole or parts (high level)
ui_print.c	creation of the print panel
ui_printP.c	callbacks and handling of the print panel
ui_printps.c	handling of the Postscript generation
ui_remote.c	creation of the remote panel
ui_remoteP.c	callbacks and handling of the remote panel
ui_result.c	handling of the result file generation
ui_selection.c	administration of the selection
ui_setup.c	creation of the setup panel
ui_setupP.c	administration of the setup panel
ui_status.c	status information in the manager panel
ui_textP.c	text output (stdout, Log file)
ui_utilP.c	utility functions
ui_xGraphic.c	X graphic interface (low level)
ui_xWidgets.c	handling of widgets
ui.h	datatypes, definitions

Table 16.2: The source files of SNNS-XGUI, Part II

16.2 Main Program

The function `main()` in the file `ui_main_.c` performs the initialisation and creates the manager panel. It then passes the control to X with XtMainLoop(). Here is the pseudo code:

```
Initialize the application;
Initialize the selection list;
Initialize the display list;
Initialize the editor;
Initialize all global variables;
Load default configuration from the file default.cfg;
Create manager panel;
Pass control to X;
```

Loading data from the configuration file changes some global variables, therefore they must be initialized already. The X event dispatcher calls the SNNS-XGUI callback routines and event handler routines. Actions that affect the network require communication with the kernel (see also chapter 16.12). The following sections describe the reactions of the windows and widgets to different events.

16.3 Manager Panel

The manager panel consists of the info panel, a message line and the XGUI menu. The panel is created at initialization time. The callback functions in the file `ui_mainProcs.c` create all the windows and the procedure `ui_printMessage()` creates messages in the panel. SNNS-XGUI is left by pressing the `QUIT` button with the function `ui_quit()`.

If multiple displays are used, all status information is best kept in one window. To display the information in different windows would waste space on the displays. Therefore this information is collected in the manager panel.

Functions, which change a value displayed in the manager panel have to use the routine `ui_stat_displayStatus` to updated the panel. The function `ui_stat_displayStatus` displays the other status information with the help of the variables `ui_sel_numberOf-SelectedItems`, `ui_key_flags`, `currentLayer` and `ui_currentDisplayPtr`. The scanning position is pased on as a parameter. To avoid flickering, this function is called only when there is a change in information (i.e. it is not called if the mouse is dragged over the display without changing the scanning position).

Sometimes the displayed attributes have to be checked for validity. The function `ui_info_-anyUnitSelected()` checks whether the displayed source and target really exist. The function `ui_info_makeUnitInfoPanelConsistent()` updates the panel, if units or sites have been erased. It is also called by several functions of the action module.

The Info panel is the most complex of SNNS-XGUI. To ease access to the widgets of the source and target units, they are accumulated in structures (`ui_targetWidgets`, `ui_sourceWidgets`, `ui_linkWidgets`). This was used in programming the routines in ui_infoProcs.c. Each attribute is also stored in a corresponding attribute structure (ui_targetUnit, ui_sourceUnit, ui_link).

16.4 Layer Panel

The toggle button in the layer panel is initialized with the bits of the variable ui_layerStartValue. Bit 0 corresponds to layer 1, bit 1 to layer 2, etc. Because the layer panel yields a value, X has to display the panel and allow user input before the program is continued (just like it was the case with the confirmer). The result is available in the variable ui_layerReturnValue.

16.5 Graphic Windows

All data for the graphic windows is stored in a linear linked list. Valid for all these windows are the following event types: GraphicExpose, ButtonPress, ButtonRelease, KeyPress, MotionNotify, EnterNotify, LeaveNotify, MapNotify, UnmapNotify, and StructureNotify.

There is no background storage for the graphic output, because it would be too memory intensive. The drawback is a longer screen setup time which depends mainly on the number and kind of link options that are displayed.

For simplicity reasons, all displays use the same graphic context ui_gc (see [You89], p. 184 for reference).

The events for all graphic windows are processed with the three event handlers ui_KEvent() for KeyPress events, ui_can_MWEvent() for mouse and window events and ui_can_MapEvent() for window mapping events. These functions are installed for each window with XtAddEventHandler. They determine in which window the event has occurred. This information is needed to determine subnet number and grid position because of the different origins in the various displays.

Input to the windows are all keyboard events, mouse events, and window events. A window is always active, if it has last sent an EnterNotify event (the mouse has been moved into the window) and is not closed. Thereby, the editor becomes independent from the windows.

The processing of graphical output is more complicated, since some operations affect only one (e.g. a refresh; UI_LOCAL), others all open windows (e.g. a change in the network; UI_GLOBAL). A simple approach would just draw a new network in every display. This, however, would take much more time than the solution found in SNNS-XGUI, where the editor determines whether it is better to re-draw the whole network or only the updates. This approach leaves some pixels white on each update, which slowly spoils the picture.

X refreshes the displays with a call to ui_refresh after each GraphicExpose event, if the count component of the event is 0 (see also [You89], p. 118). This routine is also defined as a callback function for expose events on the generation of a display.

16.5.1 Event Handler for Mouse and Window Events

This function, located in the file ui_can_MWEvent.c, does the following:

```
SWITCH (event type)
  ButtonPress:
    get event data;
    SWITCH (mouse button):
      left button:
        remember current first position for selection of area;
        draw selection rectangle;
      middle button:
        remember current position as source unit, if there is
        a unit;
  ButtonRelease:
    get event data;
    SWITCH (mouse button):
      left button:
        delete selection rectangle;
        remember second position for area and normalize both
        coordinates;
        select or unselect area;
      middle button:
        IF info panel created THEN
          show this unit as target and remembered
          unit as source;
      right button:
        IF empty position THEN
           unselect all (in this subnet)
        ELSE
           unselect this single unit
  EnterNotify (mouse is moved into a graphic window):
    set this window as actual and show new status;
  MouseMoves (mouse is moved):
    show new status, if new grid position reached;
  MouseDragg (mouse is moved with at least one button down):
    if the left button is pressed, delete old selection box,
    draw a new one.
save position of the event in the global variables
ui_pixPosOld and ui_gridPosOld;
```

The actions to these events are defined in this function. The results affect the info panel and the status panel.

16.5.2 Event Handler for Keyboard Events

This function, in the file `ui_KEvent.c`, is implemented as a big SWITCH statement. It represents an automaton for the recognition of editor commands. The transition from one state to the following state (stored in the global variable `ui_key_currentState`) is performed at the next call to the function. Depending upon the selected mode (normal, `Mode Units`, `Mode Links`), the following state is stored in one of the global variables `ui_key_returnUnitState` or `ui_key_returnLinkState`. This means that the automata changes its behaviour according to these two variables.

On `Units Move/Copy` the user is expected to click on a target position. In the meantime the automata is on hold (`UI_STATE_GETDEST`), and signals this to the event handler for mouse and window events in the global varaible `ui_outlineActiv`. When the target is defined, this event handler defines the next state of the automata.

Some actions ask the user for a value in a popup window. Since this blocks SNNS-XGUI as a whole (with `XtPopup(widget, XtGrabExclusive)`), no additional steps have to be taken.

16.5.3 Editor Actions

The file `ui_action.c` contains all editor operations that are called exclusively by the event handlers for keyboard and pointer events. The basic structures of the functions that process selected units look very much alike. The concept is the following:

```
IF operation can not be performed THEN
  error message;
  return;

delete marks of all selected units;
WHILE there is a selected unit DO
 IF desired operation for this unit can be  performed THEN
   execute operation;
   change graphic if necessary;
 ELSE
   report (if desired)
END WHILE
keep graphic and values in info panel consistent;
Redraw marks for selected units;
```

To perform the actions `Units Copy` and `Units Move`, it first has to be checked, whether all target positions in the grid are available for the operation. Therefore, the function `ui_action_checkNewPositions()` checks, whether the target position is empty for `UI_ACTION_COPY`, or occupied by unselected units for `UI_ACTION_MOVE`. The whole operation is skipped if an error condition results from the check.

After links or units have been deleted, it is possible that information in the info panel has become invalid. A call to `ui_info_makeInfoPanelConsistent()` assures consistency in that panel.

Often it is necessary to reiterate the list of selected units within the while-loop (e.g. the function `ui_action_linksMakeClique()` generates connections between all selected units). In that case, a second while-loop is used that traverses the selection list, either from the start, or from the current element.

16.5.4 Setup Panel

All setup data is contained in a structure (`struct SetupDataType`) which is itself contained in a display structure.

```
typedef struct SetupDataType {
    Bool          showValueFlg;
    Bool          showValue;
    Bool          showTitleFlg;
    Bool          showTitle;
    Bool          showLinkFlg;
    Bool          showDirectionFlg;
    Bool          showWeightFlg;
    FlintType     linkPosTrigger, linkNegTrigger;
    FlintType     unitScaleFactor;
};
```

When the setup panel is opened, the corresponding display is current. Therefore all changes are performed in the current display `ui_currentDisplayPtr`, and all inital data is read there. If a scrollbar is manipulated, the callback function `ui_thumbed()` is called to save the value in the corresponding global variable. To set the values of `showValue` and `showTitle`, there are two callback functions that assign the value to the parameter `client_data` in each menu option. The layer value is determined by the common method of the layer popup.

The toggles, that switch the various display options on and off call the function `ui_set_toggleValues()`. This function sets the corresponding component of the setup variables in the current graphic window.

- `showValueFlg`: If this flag is set, the value that determines the size of the unit is numerically displayed below it.
- `showValue`: This number gives the displayed value and can be one of the following: `UI_ACTIVATION`, `UI_OUTPUT`, `UI_INITIAL_ACTIVATION` oder `UI_BIAS`.
- `showTitleFlg`: States whether or not the unit title should be given.
- `showTitle`: States whether the unit number (`UI_NUMBER`), or the unit name (`UI_NAME`) will be the title.
- `showLinkFlg`: Links are displayed only if this flag is set.
- `showDirectionFlg`: If this flag is set, the connections will be displayed as arrows.

- `showWeightFlg`: States, whether the link weights are displayed.
- `linkPosTrigger` and `linkNegTrigger`: For positive and negative weights, these flags hold the threshold for drawing links.
- `unitScaleFactor`: If the activation of a unit is above this limit it is displayed with maximum extension.

When the `DONE` button is pressed, the setup panel becomes invisible again. The new grid size and origin are read from the panel, and the graphic window is redrawn. There the new parameters are used, and the desired picture is created.

16.5.5 Freezing Displays

The `FREEZE` button flips a switch in the display structure which prevents the high level display routines from affecting this window. The call `ui_netCompleteRefresh(UI_LOCAL)`, necessary after a GraphicExpose event, is the only one that changes a frozen display.

16.6 List Module

The construction of a panel with a list widget is a separate module. These panels are used to select functions, IO-types and F-types. There are only three functions which must react to the various lists in a specific way: `ui_list_getFirstItem()` yields the first element of the list, `ui_list_getNextItem()` the rest, and `ui_list_setValue` handles the result. Depending upon the type of the list, the result is either written to the appropriate place in the panel (source unit or target unit), or put into the global variable `ui_list_returnName`. In any case, the variable `ui_list_returnIndex` contains the list index used last. The list is constructed with `ui_list_buildList(parentWidget, type)`, where `type` can have one of the following values:

Type	Meaning
`UI_LIST_IOTYPE`	all IO-types
`UI_LIST_FTYPE`	all defined f-types
`UI_LIST_FTYPE_NAME`	ditto (edit f-type)
`UI_LIST_ACT_FUNC`	all activation functions
`UI_LIST_FTYPE_ACT_FUNC`	ditto (edit F-Typ)
`UI_LIST_OUT_FUNC`	all output functions
`UI_LIST_FTYPE_OUT_FUNC`	ditto (edit F-Typ)
`UI_LIST_SITE_FUNC`	all site functions
`UI_LIST_SITE`	all defined sites
`UI_LIST_FTYPE_SITE`	all sites on the current F-Typ
`UI_LIST_UNIT_SITE`	all sites at the current unit
`UI_LIST_LEARN_FUNC`	all teaching functions

16.7 File Panel

After clicking `DONE`, the directory and the four file names are copied from the Ascii-TextWidgets (`ui_path`, `ui_fileNET`, `ui_filePAT`, `ui_fileCFG`, `ui_fileTXT`) into the corresponding global variables (`ui_pathname`, `ui_filenameNET`, `ui_filenamePAT`, `ui_filenameCFG`, `ui_filenameTXT`).

The file `ui_fileProcs.c` contains the functions for loading and saving of nets, patterns, configurations and texts. The structure of these functions is:

```
PROCEDURE save;
IF a file with this name already exists THEN
  IF User allows overwrite THEN
   save;
  ELSE
   abort and message;
END;

PROCEDURE load;
IF a file with this name already exists THEN
  load;
  IF error THEN message;
END;
```

16.8 Help Window

The callback function `ui_help_searchText` is called by clicking one of the buttons `MORE`, `LOOK` or `TOPICS` with a flag as `client_data`. On clicking `LOOK` or `MORE` the string is read from the X primary section, and the help text is searched for this string from the beginning. On clicking `TOPICS` the string `*␣TOPICS` is searched.

16.9 Confirmer

The file `ui_confirmer.c` consists mainly of the functions listed in [SUN86], p. 348 ff (prefix `ui_cf_`) for the construction of a confirmer package adapted to X windows. This package was enhanced only for the functions `ui_confirmOk()` and `ui_confirmYes()` to highlight the calls to the confirmer in the source code. Additionally an icon (`STOP` or exclamation mark) is displayed in the confirmer.

The intrinsics of X don't allow an interuption of XtAppMainLoop(). Therefore, a temporary event dispatch loop has to be programmed (see below). This loop is exited when a button in the confirmer is pressed, whose callback routine changes the variable `ui_cf_exit` to `TRUE`. Since XtGrabExklusive is set and the XGUI toplevel widget `ui_toplevel` is the parent of the confirmer, it blocks all events for SNNS-XGUI until it is destroyed.

```
ui_cf_exit = FALSE;
while (NOT ui_cf_exit) {
    XtAppNextEvent(ui_appContext, &event);
    (void) XtDispatchEvent(&event);
}
```

16.10 Graphic

The graphic module consists of three levels: The first level is responsible for drawing the network as a whole. The middle level draws or delets single network elements. This level uses graphic primitives of the lowest level, which make up the X graphic interface, as well as basic Xlib output functions for graphics and text.

The file `ui_netUpdate.c` contains the functions for drawing the whole net (prefix `ui_net_`). All routines in this module determine in which of the open displays to draw. Iconified or frozen displays are ignored. Depending upon parameters, the function `ui_net_updateWholeNet()` draws either all links or all units. For a complete update of the net use the function `ui_net_completeRefresh()`. This function first clears the window, then draws the links and finally the units.

The file `ui_netGraph.c` contains the functions for drawing network elements (`ui_drawUnit()`, `ui_drawLink()`). Two switches are important, one to determine whether to draw or delete the element (`UI_ERASE` or `UI_DRAW`), and one to determine whether to do the output globally (`UI_GLOBAL`) or only in the current window (`UI_LOCAL`). Units are drawn if the subnet number is the same as in the window, at least one layer of the unit is activated in the window, and the display is neither frozen nor iconified. Links are drawn if the corresponding units are visible, have the same subnet number, and have a weight that exceeds the threshold set by the trigger sliders in the setup panel.

Functions to compute grid coordinates from pixel coordinates and vice versa are located in the file `ui_utilProcs.c`. Normalizing functions are also located here. Normalization of edges of a rectangle here means exchanging the coordinates, if necessary, to place the first edge to the left and above the second (`ui_normalize_rect()`). A normalization with `ui_normalize_coord()` only puts the first coordinate left of the second.

All functions which call X graphic routines are kept in the file `ui_xGraphic.c`. Among them is a function for drawing an arrow (`ui_xDrawArrow()`), a function for drawing a box (`ui_xDrawBox()`),as well as a function to delete a black or white rectangle (`ui_xDeleteRect()`). To generate an arrow head, a triangle is rotated appropriately (see also the algorithm in [Har83]).

16.11 Selection Mechanism

All functions and global variables dealing with the selection mechanism have the prefix `ui_sel_`. All selected units are held in a linear linked list starting at the pointer `ui_sel_listPtr`. It is set to point to an element on initialization. The number of selected

bit 0	selected, if set
bit 1	failure of last action on this unit; not used
bit 3	action UNITS MOVE: Unit was already moved
bit 4	action LINKS INVERSE: Links were already turned

Table 16.3: Meaning of the Bits in the flag item of each list element for the selection.

units is stored in the globale variable `ui_sel_numberOfSelectedItems`. The total number of allocated items is kept in `ui_sel_numberOfItems`. This list is always increasing, i.e. memory is not freed by `dispose()`, but all unused elements are kept in the list, marked as such (Bit 0 in the component `flags`), and are used again when needed. New elements are attached at the beginning of the list.

```
typedef struct SelectionType {
    struct SelectionType     *nextPtr;
    struct PosType           gridPos;
    int                      subNetNo;
    int                      unitNo;
    int                      copyNo;
    int                      flags;
};
```

For an unambiguous identification of the units, their subnet number is stored in addition to their grid position. This is necessary, since units in different subnets may have the same grid positions. On processing the action `Units Copy Structure` the unit number is saved, in order to keep track of the units during the copying process. After completion of the action, the component is superfluous. Table 16.3 gives the meaning of the bits in the component `flags`.

When several windows are used, units can be selected in multiple windows. The case of one unit being displayed in more than one window has to be observed. SNNS-XGUI therefore allows selection only of units belonging to an active layer, and at least one window with the same subnet number must be open. If a unit is selected in one window, the selection mark becomes visible in all displays.

When a unit is marked selected, a cross is drawn over the unit. It has to be assured that prior to drawing or deleting, all marks are reset, and get updated afterwards with the function `ui_sel_reshowItems`. All editor functions dealing with moving or deleting units (prefix "'`ui_action_`"'), have to assure consistency of the selection list. This can be done by changing the flag, since these functions anyway have access to this element, when they process the selected unit.

16.12 Interface to the Simulator Kernel

The simulator kernel of SNNS offers a set of interface functions (see also chapter 13) which allow a one way communication with the graphic routines. There the graphical user interface acts as the master, the kernel as the slave. The functions can be assigned to the following tasks:

- read and set unit, site, and link attributes and functions
- administer f-types and sites
- change the network topology
- load and save networks and patterns
- simulator functions: update, teaching

The keys to all operations of the user interface (like adding or deleting units or links, changing attributes etc.) are always the unit number and the unit position. Therefore these two attributes always have to be unambiguous. On dealing directly with the unit numbers great caution is needed, since the kernel compresses its data structures during a save, thereby changing the unit numbers. The access via unit number, however, is much faster than via position, because a sequential search can be avoided that way.

The search is necessary, however, when the user interface has to process an event (ButtonPress, ButtonRelease, KeyPress, ...) which has occured in a window. A possible improvement would be an additional data structure which sorts all units by their grid positions. A linear linked list could, for example, be assigned to each row of the grid. The search time is then limited by the maximum number of units in a row. Unfortunately, for small nets the longer time to build up the display might outweigh the faster access time. In very large nets this mechanism could be a real improvement.

To increase the stability of the simulator, the current version always requests its data from the kernel (and not from global variables) in all situations where consistency is important. There are also data consistency checks contained in the code, before any change on the net is performed. This does not decrease system performance, since these requests are answered very fast by the kernel.

The type `bool`, used by all boolean functions of the kernel, is identical with the type `Bool`, used by all SNNS-XGUI functions. The type `Bool` is used by X windows and therefore is also used in the user interface.

```
typedef int Bool; \* User Interface *\
typedef int bool; \* Kernel *\
```

All functions in this interface have the prefix "‘`krui_`"’, to be able to identify the functions in this module. They are a part of the kernel functions.

More detailed information about the interface between XGUI and the kernel can be found in chapter 13 or in the source files `krui_typ.h` and `glob_typ.h` of the kernel. Modules of the user interface which use kernel functions must include the file `kr_ui.h`.

Chapter 17

3D-Display Implementation

This chapter gives a short overview of the implementation of the 3D visualization component. The modules for 3D-viewing can be divided into four groups:

- Modules for the data structures
- Modules for the panels
- Modules for visualisation
- Modules for drawing

Each program file has a corresponding header file declaring the exported functions. Exceptions are the files `d3_global.c` and `d3_global.h`, where the global variables are defined. All modules carry the prefix `d3_` for better distinction from the 2D XGUI functions (prefix `ui_`) and the kernel functions (prefix `krui_`).

The functions that are visible from outside a module also carry the prefix `3d_` of the 3D user interface.

17.1 Contents of the Modules

The following files contain data structures and variables of the 3D user interface:

`d3_global.h`	Global data structures and constants
`d3_global.c`	Global variables
`d3_font5x7.c`	Array for the 5×7 character set
`d3_font5x8.c`	Array for the 5×8 character set
`d3_font8x14.c`	Array for the 8×14 character set

The following files contain panels of the 3D user interface:

`d3_pannels.c`	Control panel
`d3_move.c`	Panel for stepwise rotation, scaling and translation
`d3_zvalue.c`	Panel for input of the z-value
`d3_setup.c`	Panel for the step sizes and initial values

`d3_model.c`	Panel for the kind of display
`d3_project.c`	Panel for the projection
`d3_light.c`	Panel for the illumination
`d3_units.c`	Panel for displaying values in the units
`d3_links.c`	Panel for displaying values in the links
`d3_disp.c`	Handling of the display window and the event loop
`d3_xUtils.c`	X Window assistance routine

The following files contain 3D visualization functions:

`d3_main.c`	Interface and controls of the visualisation
`d3_shade.c`	Illumination algorithm
`d3_draw.c`	High level drawing routines for units and links
`d3_anageo.c`	Matrix transformations and basic operations
`d3_lists.c`	Administration of lists

The following files contain low level drawing routines:

`d3_graph.c`	Low level drawing routines and color control
`d3_zgraph.c`	z-buffer functions
`d3_point.c`	Pixel oriented output
`d3_fonts.c`	Control of text output
`d3_dither.c`	dithering of the color values

17.2 Global Data Types and Variables

All global data types and constants are declared in the module `d3_global.h`. The basic data types are

`typedef float vector[4]`	the definition of a vector
`typedef float matrix[4][4]`	the definition of a matrix
`typedef vector cube[9]`	the corner points of a cube (unit), as well as the center

The following data type is used for the z-buffer algorithm of the unit surfaces:

```
typedef struct {
    int n;
    int mask;
    vector vert[4];
} d3_polygon_type;
```

`vert`	contains the vectors of the polygon edges
`mask`	contains the interpolation mask for the x-, y- and z-coordinates
`n`	used as a counter of polygon edges

The following data type is used for the values that can be represented by a unit:

```
typedef struct {
     int size;
     int color;
     int top_label;
     int bottom_label;
} d3_unit_mode_type;
```

`size`	a value is displayed by the size of the unit
`color`	a value is displayed by the color of the unit
`top_label`	a value is written to the upper right corner of the unit
`bottom_label`	a value is written to the lower right corner of the unit

The fields may each contain one of the constants: `activation_on`, `init_act_on`, `output_on`, `bias_on`, `name_on`, `number_on`, `zvalue_on` or `nothing_on`.

The light source is described by the following declaration:

```
typedef struct {
    int    shade_mode;
    vector position;
    float  Ia, Ka, Ip, Kd;
} d3_light_type;
```

`shade_mode`	specifies, whether the units are to be illuminated
`position`	is the position of the light source
`Ia, Ka, Ip, Kd`	are the illumination constants

The whole status of the network display is contained in:

```
typedef struct {
    vector trans_vec, rot_vec, scale_vec;
    vector trans_step, rot_step, scale_step;
    vector viewpoint;
    float unit_aspect;
    float link_scale;
    float pos_link_trigger;
    float neg_link_trigger;
    int font;
    int projection_mode;
    int color_mode;
    int link_mode;
    d3_unit_mode_type unit_mode;
    d3_light_type light;
} d3_state_type;
```

`trans_vec`	current translation
`rot_vec`	current rotation
`scale_vec`	current scaling
`trans_step`	step size for the translation in the transformation panel
`rot_step`	step size for the rotation in the transformation panel
`scale_step`	step size for the scaling in the transformation panel
`viewpoint`	viewpoint for the central projection
`unit_aspect`	ratio between the edge length of the units and the distance between them
`link_scale`	specifies a factor for a region in which the links are to be drawn
`font`	specifies the character set for labeling the units and links
`projection_mode`	contains the constant `parallel` or `central`
`model_mode`	specifies the display mode for the network. The field contains the constant `wire_frame` or `solid`
`color_mode`	specifies whether a monochrome or color terminal is used
`link_mode`	contains the display mode of the links. Values may be `links_on, links_off, links_color` or `links_label`
`unit_mode`	see `d3_unit_mode_type`
`light`	see `d3_light_type`

The global variables are located in the module `d3_global.c`. Most of them are initialized with default values.

The variables

```
Display *d3_display;
Window  d3_window;
GC      d3_gc;
int     d3_screen;
```

address the display window of Xlib (the data types `Display, Window` and `GC` are declared in Xlib).

```
int d3_displayXsize;
int d3_displayYsize;
```

contain the current size of the display window.

```
bool d3_displayIsReady;
bool d3_controlIsReady;
```

specify, whether the display window and the control panel are created.

```
bool d3_freeze;
```

specifies, whether updates are possible in the display window.

```
int d3_fontXsize;
int d3_fontYsize;
```

specify the size of the current character set used to label the units and links.

```
int d3_numberWidth;
int d3_shortNumberWidth;
```

specify the width of float and short int in the X Window dialogue widgets.

```
struct TransTable *d3_xyTransTable;
```

is a pointer to the 2D → 3D translation table. The table itself is declared in the kernel.

```
int d3_transOffset;
```

specifies the offset for zero in the table. This is necessary, since C does not allow negative indices.

```
d3_state_type d3_state;
```

contains the status of the network display (see `d3_state_type`)

```
cube d3_e_cube;
```

contains the corner points of the standard cube. The cube is moved by the vector (-0.5, -0.5, -0.5) to be centered in the coordinate system.

```
int d3_cube_lines[12][2];
```

contains the twelve edges of a cube. The numbers are used as indices to `d3_e_cube`.

```
int d3_vertex_index[6][4];
```

contains the six planes of a cube. The numbers are also used as indices to `d3_e_cube`

17.3 Drawing the Network in 3D

This section describes the functions that are called to draw the network. Most of the routines are located in the module `main.c`. In other cases, the name of the module is given in parentheses without the prefix d3_ . The functions are explained "top-down". Figure 17.1 shows the hierarchy of the calls.

d3_drawNet:The routine is called by the function `ui_net_updateWhole` in the module `ui_netUpdate.c` as well as by all routines which make a redraw of the network necessary.(e.g. upon changing from wire to solid model). When the display is frozen or not opened, the routine is left immediately. In the case of solid representation, the z-buffer is cleared. Then the units are drawn by `draw_units`. If links are to be displayed, `draw_links` calls the appropriate functions.

draw_units: This function draws all units in a network. First, the maximal dimensions of the net are calculated by `get_net_extrema`, then the center of rotation is computed from there. Afterwards the matrices for the global translation, scaling, and rotation are determined. The corners for labeling are determined next by `get_label_vert_indices`. Now

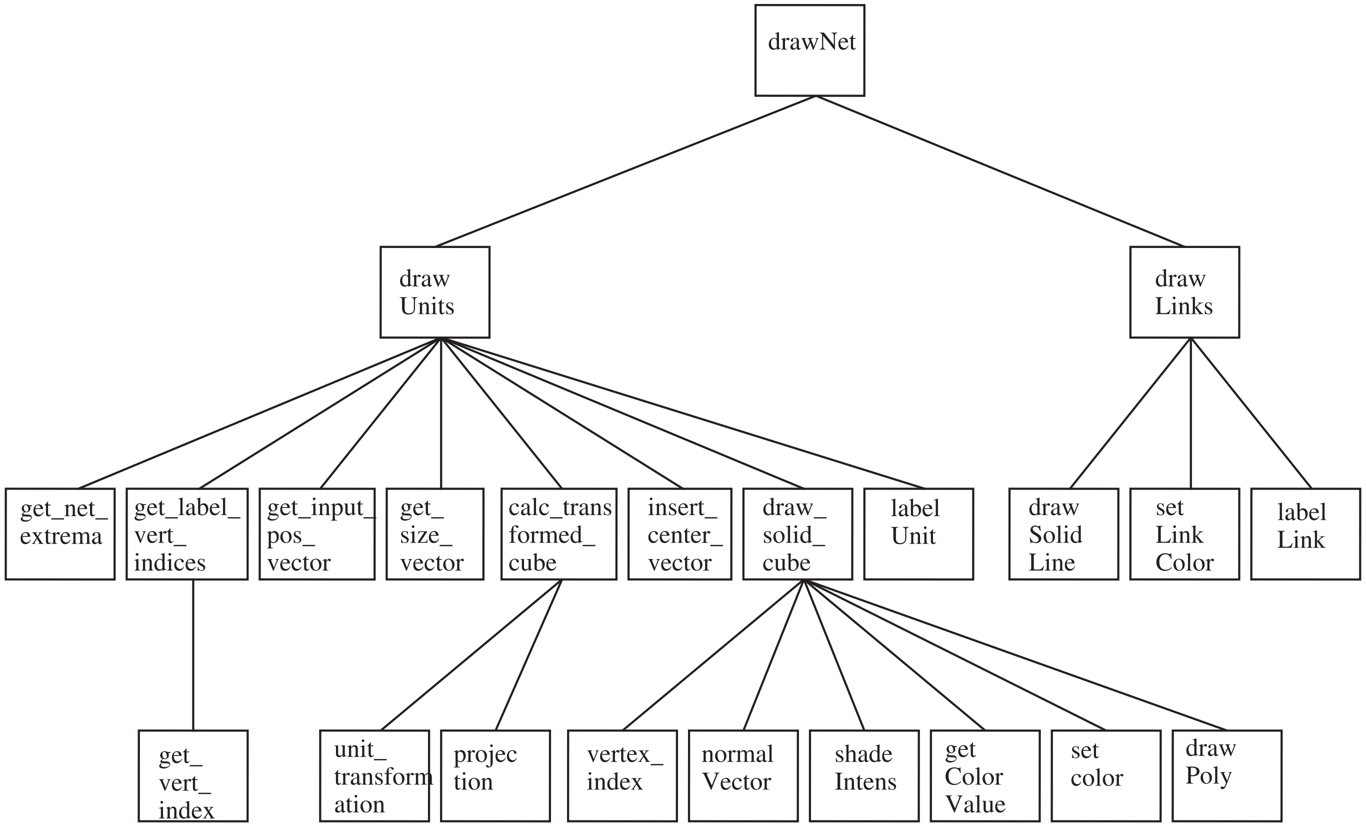

Figure 17.1: Function calls of drawNet

the net is traversed unit by unit. For each unit its location in space (in `get_unit_pos_vector`) and the corresponding matrix is computed. If a value is to be displayed by the size of the unit, the value is first changed to a vector in `get_size_vector` and then to a matrix. The matrices are multiplied with each other by `calc_transformed_cube`. With this new matrix, the vectors of the standard cube are transformed to picture space. The center of the transformed unit is stored by `insert_center_vector`, to speed up the subsequent drawing of links. Depending upon the representation mode the unit is drawn with `d3_drawWireframeCube` or `d3_drawSolidCube` in the module `draw.c`. If the units are to be labeled, `d3_labelUnit` puts the value at the appropriate place.

get_net_extrema: This routine determines the extension of the network in x-, y-, and z-direction. For each unit position it is checked whether one of its components is a maximum. The results are two vectors enclosing the net.

get_label_vert_indices: The indices to the unit vectors for labeling are computed in this function. This can be done at a single unit, since all units are rotated by the same angle and the alignment doesn't change. The routine calls the actual computing function `get_vert_index` once for the upper and once for the lower label.

get_unit_pos_vector: This function contains the 2D → 3D transformation. Input parameter is the unit number, output value the 3D–vector. All functions which require 3D–coordinates have to call this routine.

The position of the unit is read by `krui_getUnitPosition`. This value contains the x and y coordinate in the 2D–display as well as the z-value assigned to the unit. The displacement of the unit against the 2d–display in x and y direction is computed by the 2D → 3D translation table `d3_xyTransTable`. The z-coordinate is used to index the table.

get_size_vector: In this routine the activation, the initial activation, the output, or the threshold is translated into a vector for the unit size. All components of the vector contain the same value. Values bigger than 1 are clipped. This guarantees that the unit can not grow arbitrarily and cover other units. For negative values and values around zero, a flag is returned that keeps the unit from being drawn. This is necessary, since the unit would otherwise be drawn as a point.

calc_transformed_cube: This subprogram has to calculate the transformed unit coordinates. Therefore the following matrices have to be multiplied: matrix for 3D–position, for centering the unit in space, for the base values of the unit, for the size of the unit representing a value, for the scaling of the net, for a displacement of the net, and the matrix for rotation of the net. The resulting matrix transforms the vectors of the standard cube by `unit_transformation`. Finally a central projection is performed by `d3_projection` if that option was selected.

unit_transformation: In this routine, the vectors of the standard cube are multiplied by the transformation matrix.

insert_center_vector: The transformed center of the unit is written to the unit structure of the kernel by `krui_setUnitCenters`.

d3_drawWireframeCube: This function draws a "transparent" unit. The eight transformed vectors of the unit corners are connected with each other according to the entries in the `d3_cube_lines` array. The drawing is performed by the low level routine `d3_drawLine` (`graph.c`)

d3_drawSolidCube: This subprogram draws a "massive" unit. The corner points of the planes are defined by the array `d3_vertex_index` in `global.c`. If no value is to be displayed by the color of the unit, the brightness is computed for all six planes of the cube. For this purpose, `d3_normalVector` (`anageo.c`) computes a vector perpendicular to the plane. `d3_shadeIntens` in module `shade.c` takes the position of the light source, the normal vector of the plane, and a vector on the plane and computes the intensity value for the plane. This is converted to a palette index by `d3_intens_to_grayval` and activated by `d3_setColor` (`draw.c`). If a value is to be displayed by the color, `d3_getColorValue` determines the appropriate color value and `d3_value_to_color` the palette index. Then `d3_drawPoly` (`graph.c`) draws the plane.

d3_shadeIntens: Here the intensity of the light source is computed. The value varies between -1 and 1.

d3_getColorValue: This function returns a value in the range $[-1, 1]$. The value may represent the activation, initial activation, output, or threshold of the unit.

d3_labelUnit: To label a unit, the appropriate value is converted to a string. Then, the position for the string is computed. The function `get_label_vert_indices` did already compute the index of the corner. If the label is to be put at the upper corner, the y-coordinate has to be adjusted by the font height. The string is drawn by a call to `d3_draw_string` in module `fonts.c`.

draw_links: This function draws the links in a network. For each unit the numbers of all units connected with the current one are obtained by `krui_getFirstPredUnit` and `krui_getNextPredUnit`. Additionally the weight of the connection is stored. The coordinates of the units to be connected are determined by `krui_getUnitCenters`. A link is either black, or gets a color assigned by `d3_setLinkColor` which corresponds to its weight. The drawing is performed by the functions `d3_drawWireframeLine` or `d3_drawSolidLine` in module `draw.c`. An optional labeling with the weight is performed by a call to `d3_labelLink`.

d3_setLinkColor: The color of a link is determined by its weight. For this purpose the weight is converted to the interval $[-1, 1]$ with the scale factor `d3_state.link_scale`. Bigger weights get the value 1, smaller ones -1. The implementation is analogous to the 2D–display.

d3_drawWireframeLine: To draw a link in the wire frame model, the vector is rounded and drawn by the function `d3_drawLine` (`graph.c`).

d3_drawSolidLine: The points of a link in solid model display are computed from the line equation and drawn with `d3_putPixel`.

d3_labelLink: This routine writes the weight of a link at the center of the link. The center between head and tail is given by the line equation with $\mu = 0.5$. The value is put at this position with `d3_draw_string`.

17.4 Low Level Drawing Routines

The low level drawing routines are located in the modules `d3_graph.c`, `d3_fonts.c` `d3_point.c`, and `d3_dither.c`.

The following functions only call the corresponding Xlib routines. The justification for these additional routines is, that they simplify porting to other graphic systems, since they are the lowest level of drawing calls.

function	Xlib Call	Task
`d3_setColor`	`XsetForegroand`	sets the foreground color
`d3_clearDisplay`	`XClearWindow`	deletes the display window
`d3_drawLine`	`XDrawLine`	draws a line
`d3_putPixel`	`XDrawPoint`	draws a pixel

The z-buffer is controlled by the routines in module `d3_zGraph.c`. The z-buffer is organized as a one dimensional array of float. Thereby, clearing the buffer is very fast. The necessary memory is dynamically allocated (and freed) by the operating system. The position (x, y) is computed explicitly from the size of the buffer.

`d3_initZbuffer`	allocate the z-buffer on the heap
`d3_clearZbuffer`	reset the z-buffer to the maximum distance
`d3_readZbuffer`	get distance z at position (x, y)
`d3_writeZbuffer`	set distance z at position (x, y)
`d3_freeZbuffer`	free the z-buffer on the heap

A polygon is drawn by the function `d3_drawPoly`. The corresponding Xlib call cannot be used, since the visibility of each polygon point has to be checked. The implementation is derived from the article "Generic Convex Polygon Scan Conversion and Clipping" by Paul Heckbert.

Several routines for labeling the links and units had to be created. This was necessary because the characters must be addressed point by point because of the z-buffering. The routines are listed in the module `d3_fonts.c`.

`d3_select_font`	selects a font
`d3_get_font_sizes`	returns the width and height of the current font
`d3_draw_string`	writes the string to the position (x, y, z)

The local functions `draw_char` and `draw_zbuffered_char` are used by `d3_draw_string` to generate a pixel sequence from the ASCII code of a character and the font array.

The creation of grey values is performed with the function `dither` in module `d3_dither.c`.

17.5 Matrix Calculations

The module `d3_anageo.c` contains the routines for vector and matrix calculations.

`e_matrix`	creates an identity matrix
`d3_transMatrix`	creates the translation matrix
`d3_scaleMatrix`	creates the scaling matrix
`d3_rotateXmatrix`	creates matrix for rotation around the x-axis
`d3_rotateYmatrix`	creates matrix for rotation around the y-axis
`d3_rotateZmatrix`	creates matrix for rotation around the z-axis
`d3_rotateMatrix`	creates matrix for rotation around all axes
`d3_multMatrix`	multiplies two matrices
`d3_multMatrixVector`	multiplies a vector and a matrix
`d3_normalVector`	calculates the normal vector for a plane
`d3_projection`	performs a central projection

17.6 The 3D Display Window

The 3D display window is opened by the callback function `d3_createDisplayWindow`, linked to the `DISPLAY` button in the control panel.

The display window has its own event handler. It reacts to ConfigureNotify and Expose events. When an Expose event occurs, the net is redrawn. Since a series of Expose events is created in some cases, all but the last one are ignored. If the size of the display window is changed, a ConfigureNotify event causes the net to be centered in the new window and redrawn. All Expose events are blocked in this case.

The display window is created by `XtCreatePopupShell`. It contains an inner frame (see 17.7). After the window becomes visible the Xlib variables `d3_display`, `d3_window` and `d3_screen` are created as well as the graphic context `d3_gc`. Each subsequent reference to the display window is performed by Xlib functions (see 17.4).

17.7 Panels

The panels were implemented with the Intrinsics – and Athena Toolkit for X11 Release 4. The structure of the panels is similar to those of the 2D–interface, to make the display as homogeneous as possible. Therefore, the routines in module `ui_xWigets.c` have been used.

The module `d3_xUtils.c` contains the two functions `d3_xCreateButtonItem` and `d3_xCreateToggleItem`. They create the buttons and import the "xbm" bitmaps for labeling the buttons.

Most of the panels are programmed in a similar fashion. Therefore, the implementation of the panel for the projection is described as an example. The panel looks like this:

In module `d3_panels.c` five functions handle the project panel.

d3_createProjectpanel: This routine is a callback function of the button PROJECT in the control panel. As a parameter, the button itself is passed. From there, `XtTranslateCoords` computes the position at which the project panel is to be opened with `Xt-`

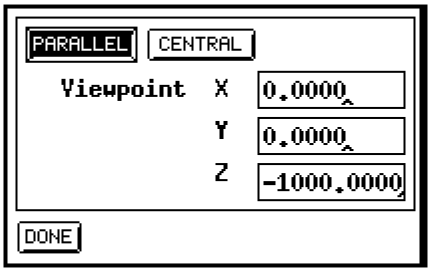

Figure 17.2: Project panel

`CreatePopupShell`. Since the panel is supposed to have an inner frame, a window of `boxWidgetClass` is created with `XtCreateManagedWidget`. There the actual window of class `formWidgetClass` is located. Now the two radio buttons PARALLEL and CENTRAL are created with `d3_xCreateToggleItem`. Then the labeling **Viewpoint X** with `ui_xCreateLabelItem` and the dialogue window follow. The next two lines are created similarly. Then the DONE button is created with `d3_xCreateButtonItem`. In the next step the three callbacks for the three buttons are created by `XtAddCallback`. `setParallelProjection` and `setCentralProjection` are linked to the two radio buttons, `d3_closeProjectpanel` to the DONE button. Now a temporary variable `temp_state` of type `d3_state_type` for the status of the display is generated. All changes to the status are stored there as long as the panel exists. Now one of the two radio buttons gets inverted by `setProjectToggleState`, corresponding with the current status. At last, the panel is made visible by `XtPopup` with `XtGrabExclusive`.

setParallelProjection: If the PARALLEL button is clicked, this function sets the field `temp_state.projection_mode` to `parallel`. A call to `setProjectToggleState` updates the panel.

setCentralProjection: like `setParallelProjection`

setProjectToggleState: With this function, the status of the two radio buttons is updated. The variable `temp_state.projection_mode` is read, one of the buttons is set, the other reset.

d3_closeProjectpanel: When the panel is closed, the values of the dialogue windows have to be written to the variable `temp_state.viewpoint`. Now `temp_state` is compared with the initial status variable `d3_state`. If they differ, a parameter has changed and the net has to be redrawn. This is done by copying `temp_state` to `d3_state` and calling `d3_drawNet` afterwards. The indirection with the temporary variable is necessary, since the panel may have been opened by mistake or for control reasons. In this case the net doesn't have to be redrawn.

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 */


/* nine 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 :"
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


/* 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_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
```

Appendix B

Example Network Files

The lines in the connection definition section have been truncated to 80 characters per line for printing purposes.

B.0.4 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.0.5 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 V3.0 Batchlearning Program
Configuration file: 'newbat.config'
Log file          : 'test.log2'

Networkfile '/usr/local/bv/SNNS/SNNSv3.0/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/SNNSv3.0/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 V3.0 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 V3.0 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

[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.

[DH73] R. Duda and P. Hart. *Pattern Classification and Scene Analysis.* Wiley & Sons, Inc, 1973.

[Elm89] J. Elman. Finding structure in time. Technical Report CRL Tech Report 8801, University of California at San Diego, Center for Research in Language., 1989.

[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.

[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.

[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.

[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.

[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.

[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.

[RM86] D.E. Rumelhart and J.L. McClelland. *Parallel Distributed Processing*, volume 1. MIT Press, 1986.

[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.

[Sie91] J. Sienel. Kompensation von Störgeräuschen in Spracherkennungssystemen mittels neuronaler Netze. Studienarbeit 1037, IPVR, Universität Stuttgart, 1991.

[SK92] J. Sch"urmann and U. Kre"sel. Mustererkennung mit statistischen methoden. Technical report, Daimler-Benz AG, Forschungszentrum Ulm, Institut f"ur Informatik, 1992.

[Som89] T. Sommer. Entwurf und Realisierung einer graphischen Benutzeroberfläche für einen Simulator konnektionistischer Netzwerke. Studienarbeit 746, IPVR, Universität Stuttgart, 1989.

[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.

[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.

[You89] D.A. Young. *X-Window-System - Programming and Applications with Xt.* Prentice Hall, 1989.

[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