UNIVERSITY OF STUTTGART

INSTITUTE FOR PARALLEL AND DISTRIBUTED HIGH PERFORMANCE SYSTEMS (IPVR)

Applied Computer Science – Image Understanding

SNNS

Stuttgart Neural Network Simulator

User Manual, Version 4.0

UNIVERSITY OF STUTTGART

INSTITUTE FOR PARALLEL AND DISTRIBUTED HIGH PERFORMANCE SYSTEMS (IPVR)

Applied Computer Science – Image Understanding

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

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

SNNS

Stuttgart Neural Network Simulator

User Manual, Version 4.0

Report No. 6/95

All Rights reserved

Contents

Chapter 1

Introduction to SNNS

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

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

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

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

[1]X Graphical User Interface

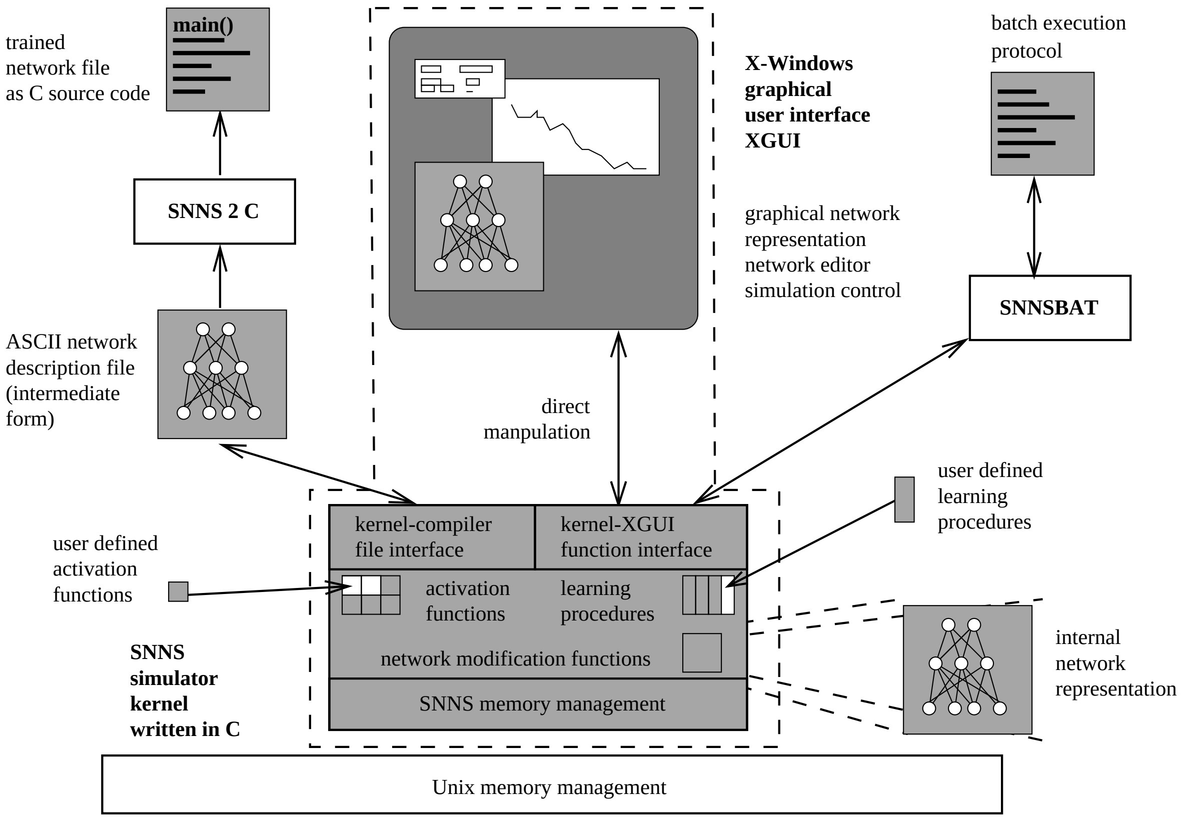

Figure 1.1: SNNS components: simulator kernel, graphical user interface xgui, snnsbat, and network compiler snns2c

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

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

This document is structured as follows:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 2

Licensing, Copying and Acknowledgments

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

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

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

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

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

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

2.1 SNNS License

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

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

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

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

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

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

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

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

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

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

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

2.2 How to obtain SNNS

The SNNS simulator can be obtained via anonymous ftp from host

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

in the subdirectory

```
/pub/SNNS
```

as file

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

or in several parts as files

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

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

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

and then use the command

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

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

2.3 Installation

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

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

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

```
configure
```

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

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

```
build sim
```

or with the two commands

```
build kernel
build xgui
```

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

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

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

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

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

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

The simulator is then called simply with

```
snns
```

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

```
build all
```

which calls the three commands

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

Important Note:

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

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

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

2.4 Acknowledgments

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

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

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

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

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

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

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

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

Martin Riedmiller, University of Karlsruhe
Implementation of RPROP in SNNS

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

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

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

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

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

Randolf Werner, University of Koblenz
Support for NeXT systems

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

Michael Berthold, University of Karlsruhe
Implementation of enhanced RBF algorithms

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

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

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

2.5 New Features of Release 4.0

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

New Features of Release 4.0:

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

Chapter 3

Neural Network Terminology

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

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

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

3.1 Building Blocks of Neural Nets

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

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

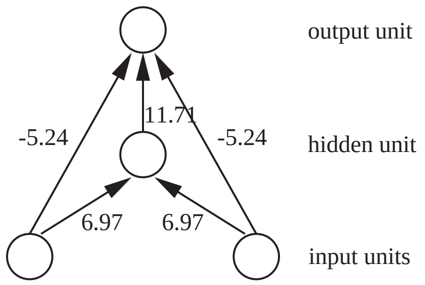

Figure 3.1: A small network with three layers of units

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

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

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

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

3.1.1 Units

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

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

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

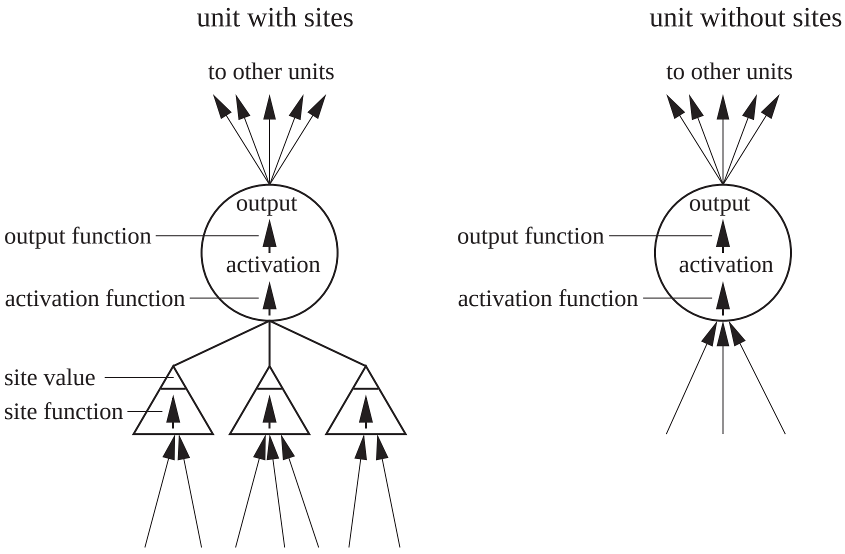

Figure 3.2: One unit with sites and one without

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

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

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

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

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

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

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

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

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

where:

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

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

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

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

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

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

This yields the well-known logistic activation function

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

where:

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

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

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

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

where:

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

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

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

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

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

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

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

3.1.2 Connections (Links)

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

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

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

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

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

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

3.1.3 Sites

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

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

3.2 Update Modes

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

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

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

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

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

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

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

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

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

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

3.3 Learning in Neural Nets

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

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

where:

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

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

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

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

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

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

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

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

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

where:

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

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

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

3.4 Generalization of Neural Networks

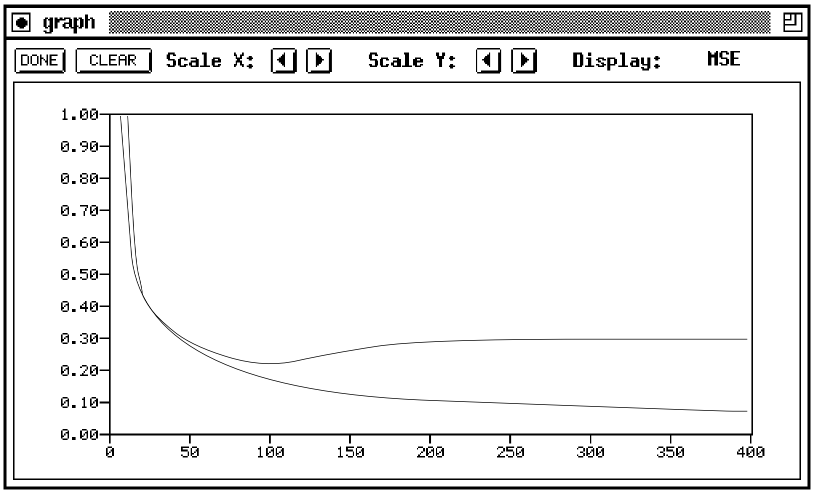

Figure 3.3: Error development of a training and a validation set

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

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

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

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

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

3.5 An Example of a simple Network

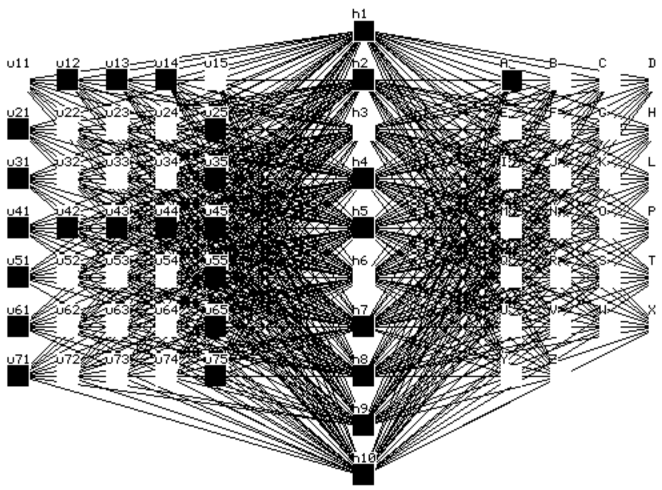

Figure 3.4: Example network of the letter classifier

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

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

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

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

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

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

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

Chapter 4

Using the Graphical User Interface

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

4.1 XGUI Files

The graphical user interface consists of the following files:

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

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

```
xgui/bin/<architecture>/xgui
```

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

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

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

`setenv XGUILOADPATH` *Path*

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

SNNS uses the following extensions for its files:

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

A simulator run is started by the command

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

where valid options are

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

in the home directory of SNNS or by directly calling

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

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

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

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

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

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

4.2 Windows of XGUI

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

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

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

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

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

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

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

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

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

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

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