UNIVERSITY OF STUTTGART

INSTITUTE FOR PARALLEL AND DISTRIBUTED HIGH PERFORMANCE SYSTEMS (IPVR)

Applied Computer Science – Image Understanding

SNNS

Stuttgart Neural Network Simulator

Manual Extensions of 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

Extensions of Version 3.3

Contents

Chapter 1

The Purpose of this Document

There are roughly two new releases of SNNS every year. At this pace it is clear that the changes from version to version are not fundamental and that the corresponding changes and additions in the documentation affect only a few pages of the manual.

In order to save a few trees it may be advisable to not print out all the documentation, that actually hasn't changed much, again and again. So we decided to provide a sort of an update document to the manual. This document is prepared for all those users that already have an older manual and wish to receive only an update for the new functionality.

We do like to point out, however, that this document can only contain those parts that are entirely new. All those little improvements, cross references, and additions in chapters that are mainly unchanged can not be included here. So we encourage the users either to use the complete manual online or at least to print out every other release of the manual in order to become aware of these changes.

Beginners anyways should start with the full documentation and ignore these excerpts.

1.1 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 alternated 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).

4.2.3 Manager Panel

Figure 4.1 shows the manager panel. From the manager panel all other elements that have a different, independent window assigned can be called. Because this window is of such central importance, it is recommended to keep it visible all the time.

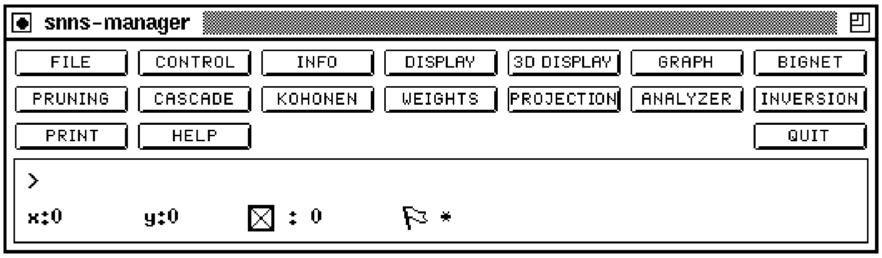

Figure 4.1: Manager panel

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

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

SNNS Status Message:

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

Status line

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

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

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

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

4.2.4 Info Panel

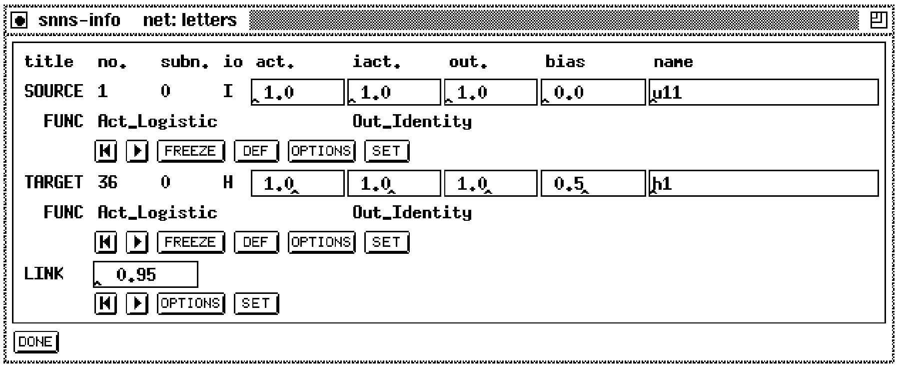

Figure 4.2: Info panel

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`.

In table 4.1 the various fields are listed. The fields in the second line of the SOURCE or TARGET unit display the name of the activation function, name of the output function, name of the f-type (if available). The fields in the line `LINK` have the following meaning: weight, site value, site function, name of the site. Most often only a link weight is available. In this case no information about sites is displayed.

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

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

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

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

1. Arrow button |◀: The button below TARGET selects the first target unit (of the given source unit); the button below SOURCE selects the first source unit (of the given target unit);
2. Arrow button ▶: The button below TARGET selects the next target unit (of the given source unit); the button below SOURCE selects the next source unit (of the given target unit);
3. `FREEZE`: Unit is frozen, if this button is inverted. Changes become active only after `SET` is clicked.
4. `DEF`: The default unit is assigned the displayed values of `TARGET` and `SOURCE` (only activation, bias, IO-type, subnet number, layer numbers, activation function and output function).
5. `OPTIONS`: Calls the following menu:

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

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

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

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

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

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

4.4 Update Functions

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

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

ART1_stable	for ART1 networks
ART1_synchronous	for ART1 networks
ART2_Stable	for ART2 networks
ART2_Synchronous	for ART2 networks
ARTMAP_Stable	for ARTMAP networks
ARTMAP_Synchronous	for ARTMAP networks
Auto_Synchronous	for Autoassociative Memory networks
BAM_Order	for Bidirectional Associative Memory networks
BBTT_Order	for Backpropagation-Through-Time networks
CC_Order	for Cascade- Correlation networks

CounterPropagation	for Counterpropagation networks
Dynamic_LVQ	for Dynamic-Learning-Vector-Quantization networks
Hopfield_Fixed_Act	for Hopfield networks
Hopfield_Synchronous	for Hopfield networks
JE_Order	for Jordan or Elman network
JE_Special	for Jordan or Elman network
Kohonen_Oder	for Self-Organizing Maps (SOMS)
RCC_Order	for reccurrent-Cascade-Correlation networks
Random_Order	for any network
Random_Permutation	for any network
Serial_Order	for any network
Synchronous_Order	for any network
TimeDelay_Order	for Time-Delay networks
Topological_Order	for any network

All these functions receive their input from the five update parameter fields in the control panel. See figure 4.16

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

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

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

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

ART1_Stable

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

ART1_Synchronous

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

ART2_Stable

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

ART2_Synchronous

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

ARTMAP_Stable

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

ARTMAP_Synchronous

The first step is to calculate the output value of the input units (input units of ARTa, ARTb). Now a complete propagation step takes place, i.e. all units calculate their output and activation value. The search for two recognition neuron with highest activation follows.

The search takes place in both ARTa and ARTb. The required parameters are ρ^a, ρ^b, ρ^c in field1, field2, field3 respectively.

Auto_Synchronous

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

BAM_Order

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

BBTT_Order

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

CC_Order

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

CounterPropagation

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

Dynamic_LVQ

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

Hopfield_Fixed_Act

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

Hopfield_Synchronous

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

JE_Order

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

JE_Special

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

Kohonen_Order

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

RCC_Order

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

Random_Order

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

Random_Permutation

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

Serial_Order

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

Synchronous_Order

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

TimeDelay_Order

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

Topological_Order

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

4.5 Initialization Functions

In order to work with various net-models and learning algorithms, different initialization functions that initialize the components of a net are required. Backpropagation for example will not as long as all weights have the same output value, since there can be no gradient computed. Click `OPTIONS` in the control panel to select an init function.

The following initialization functions are available:

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

All these functions receive their input from the five fields in the initialization panel. See figure 4.16.

Here is a small decription of the different initialization functions:

ART1_Weights

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

ART2_Weights

For an ART2 network the weights of the top-down-links ($F_2 \rightarrow F_1$ links) are set to 0.0 according to the theory ([CG87]). The choice of the initial bottom-up-weights is decribed in chapter 9.12.2.2.

ARTMAP_Weights

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

CC_Weights

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

ClippHebb

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

CPN_Rand_Pat

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

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

CPN_Weights_v3.2

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

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

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

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

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

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

CPN_Weights_v3.3

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

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

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

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

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

DLVQ_Weights

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

Hebb

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

Hebb_FixAct

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

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

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

JE_Weights

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

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

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

Kohonen_Weights_v3.2

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

Kohonen_Const

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

Kohonen_Rand_Pat

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

PseudoInv

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

Randomize_Weights

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

Random_Weights_Perc

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

RBF_Weights

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

RBF_Weights_Kohonen

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

RBF_Weights_Redo

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

RCC_Weights

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

RM_Random_Weights

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

9.8.2 Pruned-Cascade-Correlation (PCC)

9.8.2.1 The Algorithm

The aim of Pruned-Cascade-Correlation (PCC) is to minimize the *expected* test set error, instead of the actual training error [Weh94]. PCC tries to determine the optimal number of hidden units and to remove unneeded weights after a new hidden unit is installed. As pointed out by Wehrfritz, selection criteria or a hold-out set, as it is used in "stopped-learning", may be applied to digest away unneeded weights. In this release of SNNS, however, only selection criteria for linear models are implemented.

The algorithm works as follows (CC steps are printed italic):

1. *Train the connections to the output layer*
2. Compute the selection criterion
3. *Train the candidates*
4. *Install the new hidden neuron*

5. Compute the selection criterion

6. Set each weight of the last inserted unit to zero and compute the selection criterion; if there exists a weight, whose removal would decrease the selection criterion, remove the link, which decreases the selection criterion **most**. Goto step 5 until a further removal would increase the selection criterion.

7. Compute the selection criterion; if it is greater than the one, computed before inserting the new hidden unit, notify the user that the net is getting too big.

9.8.2.2 Mathematical Background

In this release of SNNS, three model selection criteria are implemented: the Schwarz's Bayesian criterion (SBC), Akaikes information criterion (AIC) and the conservative mean square error of prediction (CMSEP). The *SBC*, the default criterion, is more conservative compared to the *AIC*. Thus, pruning via the *SBC* will produce smaller networks than pruning via the *AIC*. Be aware that both *SBC* and *AIC* are selection criteria for *linear* models, whereas the *CMSEP* does not rely on any statistical theory, but happens to work pretty well in an application. These selection criteria for linear model can sometimes directly be applied to nonlinear models, if the sample size is large.

9.11 Dynamic Decay Adjustment for RBFs (RBF–DDA)

9.11.1 The Dynamic Decay Adjustment Algorithm

The Dynamic Decay Adjustment (DDA–)Algorithm is an extension of the RCE-Algorithm (see [Hud92, RCE82]) and offers easy and constructive training for Radial Basis Function Networks. RBFs trained with the DDA-Algorithm often achieve classification accuracy comparable to Multi Layer Perceptrons (MLPs)[1] but training is significantly faster ([BD95]).

An RBF trained with the DDA-Algorithm (RBF-DDA) is similar in structure to the common feedforward MLP with one hidden layer and without shortcut connections:

- The number of units in the **input layer** represents the dimensionality of the input space.

- The **hidden layer** contains the RBF units. Units are added in this layer during training. The input layer is fully connected to the hidden layer.

- Each unit in the **output layer** represents one possible class, resulting in an 1-of-n or binary coding. For classification a winner-takes-all approach is used, i.e. the output with the highest activation determines the class. Each hidden unit is connected to exactly one output unit.

[1] As usual the term MLP refers to a multilayer feedforward network using the scalar product as a propagation rule and sigmoids as transfer functions.

The main differences to an MLP are the activation function and propagation rule of the hidden layer: Instead of using a sigmoid or another nonlinear squashing function, RBFs use localized functions, radial Gaussians, as an activation function. In addition, a computation of the Euclidian distance to an individual reference vector replaces the scalar product used in MLPs:

$$R_i(\vec{x}) = exp(-\frac{||\vec{x} - \vec{r}_i||^2}{\sigma_i^2})$$

If the network receives vector $\vec{x}$ as an input, R_i indicates the activation of one RBF unit with reference vector $\vec{r}_i$ and standard deviation σ_i.

The output layer computes the output for each class as follows:

$$f(\vec{x}) = \sum_{i=1}^{m} A_i * R_i(\vec{x})$$

with m indicating the number of RBFs belonging to the corresponding class and A_i being the weight for each RBF.

An example of a full RBF-DDA is shown in figure 9.3. Note that there do not exist any shortcut connections between input and output units in an RBF-DDA.

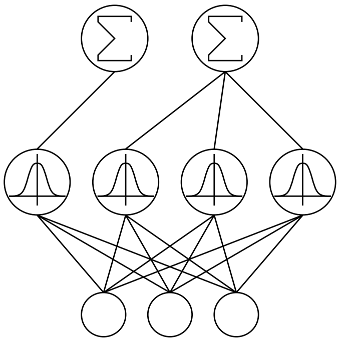

Figure 9.3: The structure of a Radial Basis Function Network.

In this illustration the weight vector that connects all input units to one hidden unit represents the centre of the Gaussian. The Euclidian distance of the input vector to this reference vector (or *prototype*) is used as an input to the Gaussian which leads to a local response; if the input vector is *close* to the prototype, the unit will have a high activation. In contrast the activation will be close to zero for larger distances. Each output unit simply computes a weighted sum of all activations of the RBF units belonging to the corresponding class.

The DDA-Algorithm introduces the idea of distinguishing between *matching* and *conflicting* neighbors in an *area of conflict*. Two thresholds θ^+ and θ^- are introduced as illustrated in figure 9.4.

Normally, θ^+ is set to be greater than θ^- which leads to a *area of conflict* where neither matching nor conflicting training patterns are allowed to lie[2]. Using these thresholds, the algorithm constructs the network dynamically and adjusts the radii individually.

[2]The only exception to this rule is the case where a pattern of the same class lies in the area of conflict but is covered by another RBF (of the correct class) with a sufficiently high activation.

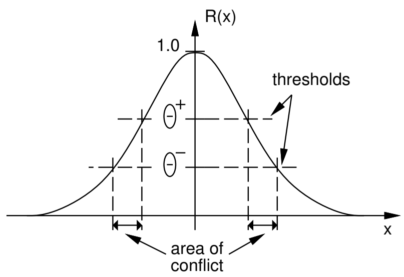

Figure 9.4: One RBF unit as used by the DDA-Algorithm. Two *thresholds* are used to define an *area of conflict* where no other prototype of a conflicting class is allowed to exist. In addition, each training pattern has to be in the inner circle of at least one prototype of the correct class.

In short the main properties of the DDA-Algorithm are:

- **constructive training**: new RBF nodes are added whenever necessary. The network is built from scratch, the number of required hidden units is determined during training. Individual radii are adjusted dynamically during training.

- **fast training**: usually about five epochs are needed to complete training, due to the constructive nature of the algorithm. End of training is clearly indicated.

- **guaranteed convergence**: the algorithm can be proven to terminate.

- **two uncritical parameters**: only the two parameters θ^+ and θ^- have to be adjusted manually. Fortunately the values of these two thresholds are not critical to determine. For all tasks that have been used so far, $\theta^+ = 0.4$ and $\theta^- = 0.2$ was a good choice.

- **guaranteed properties** of the network: it can be shown that after training has terminated, the network holds several conditions for all training patterns: wrong classifications are below a certain threshold (θ^-) and correct classifications are above another threshold (θ^+).

The DDA-Algorithm is based on two steps. During training, whenever a pattern is misclassified, either a new RBF unit with an initial weight = 1 is introduced (called *commit*) or the weight of an existing RBF (which covers the new pattern) is incremented. In both cases the radii of conflicting RBFs (RBFs belonging to the wrong class) are reduced (called *shrink*). This guarantees that each of the patterns in the training data is covered by an RBF of the correct class and none of the RBFs of a conflicting class has an inappropriate response.

Two parameters are introduced at this stage, a *positive threshold* θ^+ and a *negative threshold* θ^-. To commit a new prototype, none of the existing RBFs of the correct class has an activation above θ^+ and during shrinking no RBF of a conflicting class is allowed to have an activation above θ^-. Figure 9.5 shows an example that illustrates the first few training steps of the DDA-Algorithm.

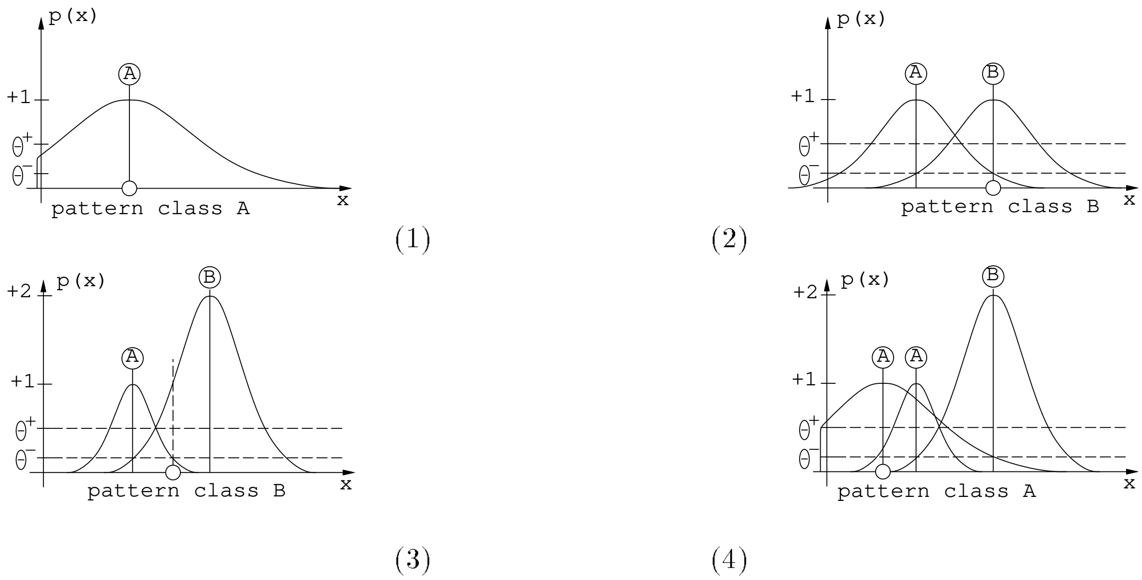

Figure 9.5: An example of the DDA-Algorithm: (1) a pattern of class A is encountered and a new RBF is created; (2) a training pattern of class B leads to a new prototype for class B and shrinks the radius of the existing RBF of class A; (3) another pattern of class B is classified correctly and shrinks again the prototype of class A; (4) a new pattern of class A introduces another prototype of that class.

After training is finished, two conditions are true for all input–output pairs[3] $(\vec{x}, c)$ of the training data:

- at least one prototype of the correct class c has an activation value greater or equal to θ^+:

$$\exists i : R_i^c(\vec{x}) \geq \theta^+$$

- all prototypes of conflicting classes have activations less or equal to θ^- (m_k indicates the number of prototypes belonging to class k):

$$\forall k \neq c, 1 \leq j \leq m_k : R_j^k(\vec{x}) \leq \theta^-$$

For all experiments conducted so far, the choice of θ^+=0.4 and θ^-=0.2 led to satisfactory results. In theory, those parameters should be dependent on the dimensionality of the feature space but in practice the values of the two thresholds seem to be uncritical. Much more important is that the input data is normalized. Due to the radial nature of RBFs each attribute should be distributed over an equivalent range. Usually normalization into [0,1] is sufficient.

[3]In this case the term "input–class pair" would be more justified, since the DDA–Algorithm trains the network to classify rather than approximate an input–output mapping.

9.11.2 Using RBF–DDA in SNNS

The implementation of the DDA-Algorithm always uses the Gaussian activation function `Act_RBF_Gaussian` in the hidden layer. All other activation and output functions are set to `Act_Identity` and `Out_Identity`, respectively.

The *Learning function* has to be set to `RBF-DDA`. No *Initialization* or *Update* functions are needed.

The algorithm takes three arguments that are set in the first three fields of the `LEARN` row in the *control panel*. These are θ^+, θ^- $(0 < \theta^- \leq \theta^+ < 1)$ and the maximum number of RBF units to be diplayed in one row. This last item allows the user to control the appearance of the network on the screen and has no influence on the performance. Specifying 0.0 leads to the default values $\theta^+ = 0.4$, $\theta^- = 0.2$ and to a maximum number of 20 RBF units displayed in a row.

Training of an RBF starts with either:

- an **empty** network, i.e. a network consisting only of input and output units. No connections between input and output units are required, hidden units will be added during training. This can easily be generated with the tool `BIGNET` (choice `FEED FORWARD`)
- a **pretrained** network already containing RBF units generated in an earlier run of `RBF-DDA` (all networks not complying with the specification of an RBF-DDA will be rejected).

After having loaded a training pattern set, a *learning-epoch* can be started by pressing the `ALL` button in the *control panel*. At the beginning of each epoch, the weights between the hidden and the output layer are automatically set to zero. Note that the resulting RBF and the number of required learning epochs can vary slightly depending on the order of the training patterns. If you train using a single pattern (by pressing the `SINGLE` button) keep in mind that every training step increments the weight between the RBF unit of the correct class covering that pattern and its corresponding output unit. The end of the training is reached when the network structure does not change any more and the Mean Square Error (MSE) stays constant from one epoch to another.

The first desired value in an output pattern that is greater than 0.0 will be assumed to represent the class this pattern belongs to; only one output may be greater than 0.0. If there is no such output, training is still executed, but no new prototype for this pattern is commited. All existing prototypes are shrunk to avoid coverage of this pattern, however. This can be an easy way to define an “error”–class without trying to model the class itself.

Chapter 12

SNNS for Workstation Clusters

Training several neural networks on a single workstation may require a large amount of training time, if performed on a single workstation. Dividing the training task among several workstations on a LAN (local area network) may significantly decrease the overall training time. This version of SNNS for workstation clusters allows to train several (possibly different) networks on different workstations and to control them with a single graphical user-interface.

The kernel interface has been extended for this purpose by an *rpc* module that is controlled by the graphical user-interface xgui. The module was named *rpc* because the communication among the workstations is performed via the rpc (remote procedure call) protocol.

This parallel SNNS version supports the following methods of parallelism:

- training different networks in parallel, each with its own pattern file,
- training several copies of the same network with different parameters or different initialization in parallel, each with its own (copied) pattern file, or
- an arbitrary combination of the above two methods.

The parallel version currently does NOT support dividing the training of a single network over several workstations. This might be added in a future version, if it can be implemented efficiently enough.

The extension consists of two programs:

- `xguirpc`: SNNS simulator program (xgui, simulator kernel, and rpc module linked into one executable program). This program is the graphical user-interface, which controls the different simulator kernels. It has the same functionality as the xgui (snns) program plus different panels to control the rpc-interface. To use multiple simulator kernels, one may start as many `kernelrpc` programs as desired.
- `kernelrpc`: simulator kernel linked with the rpc module (the server). This program is the stand alone simulator kernel with rpc module, which is controlled from the `xguirpc` program. Each kernel has is own net, pattern sets and configurations.

The files are located in the directory

<snns-root>/rpc/bin/<architecture>

where <architecture> is one of the currently supported machine architectures. We have tested it on an Intel Paragon and single CPU Sun SparcStations, and Linux systems. To use the program efficiently on multi-CPU systems like the DEC Alpha some adjustments in the libraries might be necessary.

The easiest way to start a new simulator kernel is from the **new panel** called by pressing the `NEW` button in the **SNNS rpc panel** (see 12.2.1).

If the networks are to be trained without human supervision, a similar functionality may also be obtained by starting a copy of the batch SNNS simulator `snnsbat` on each workstation.

12.1 SNNS RPC Programs

Xguirpc

`Xguirpc` has the same options as `xgui`. Additionally there is one more option:

`-force`:
The force option is instructing `xguirpc` to delete the portmap number of an old `xguirpc`. This option is only necessary if an `xguirpc` was not terminated correctly on the machine. Only the last registered `xguirpc` will receive messages from the simulator kernel. This implies that a user can run only one `xguirpc` on a computer.

Kernelrpc

`Kernelrpc` has the following options:

- `uid` <xguirpc.user-id>: the user-id of the `xguirpc` user,
- `host` <xgui_rpc_host>: the host-name of the `xguirpc`,
- `program` <xgui_program_number>: the program number under which the `xguirpc` is registered at the portmapper,
- `version` <xgui_version_number>: the version number under which the `xguirpc` is registered at the portmapper,
- `type` <type>: kernel type (normal or masters),
- `net` <net-name>: the name of a network file that is to be loaded,
- `pat` <pat-name>: the name of a pattern file that is to be loaded. This option may be given more than once.

If `kernelrpc` is started without any options the kernel will wait a short period of time. If there is no connection to an `xguirpc` the `kernelrpc` will be aborted. The default waiting time is set to 15 minutes. The simplest way to start `kernelrpc` is with the help of the **New panel** of the rpc module, because all options are set automatically. It is possible to run up to 10 `kernelrpc`s at the same time on one computer.

In some networks problems occur, caused by the domain name. If NIS (SUN Yellow pages) is activated, the function `getdomainname` does not deliver the correct domain name. However, this problem can be solved, if the environment variable DOMAINNAME is set to the correct domain name.

`Example:` On our Sun workstations `getdomainname` returns `infra` instead of `informatik.uni-stuttgart.de`. To solve this problem, we type: `setenv DOMAINNAME informatik.uni-stuttgart.de`. The best way to achieve this is to add the above line to the `.cshrc` file.

12.2 Control Panels

The distribution of SNNS onto the workstation cluster and the distributed execution of the training and recall is controlled via several windows. The main control panel, the **rpc panel**, is opened by clicking the rpc button in the manager panel.

12.2.1 SNNS RPC Panel

This panel is the control center of the communication module. In this panel the simulator kernel with which the graphical user-interface will interact is selected. Choose a row and click `SWITCH TO`, to specify a simulator kernel to contact. The displays and the control panel will then be updated.

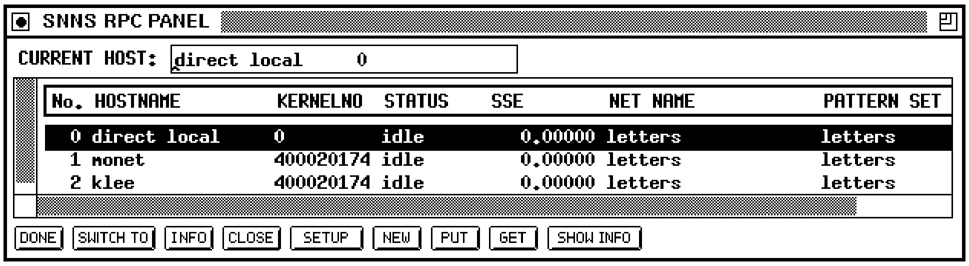

Figure 12.1: The RPC Panel

With this panel the current simulator kernel, shown in the top row, is chosen.

The buttons and fields of this panel (shown in figure 12.1) have the following functionality:

1. **CURRENT HOST** gives the name of the host computer on which the currently selected `kernelrpc` program is running.
2. The **list window** displays the list of all currently registered `kernelrpc` programs.

3. `SWITCH TO` switches to the selected simulator kernel.
4. `INFO` displays information about the selected simulator kernel. No switch of the simulator kernel takes place. The information displyed includes: Network names or comments which are saved in the network description.
5. `CLOSE` closes the selected simulator kernel. Before this process is completed, the system switches to 'direct local'. This simulator kernel is special because it is not using the standard communication protocol. It is impossible to delete this kernel because it is linked directly to the graphical user-interface. If the graphical user-interface is started, the standard simulation kernel is 'direct local'. The only difference between a 'direct local' rpc kernel and the standard SNNS kernel without rpc interface is the existence of the `RPC` button in the **Manager panel** with the added rpc functionality.
6. `SETUP` opens the **Setup panel**. In this panel the current settings of the marked kernel are displayed and can be modified. These settings are described in section 12.2.2.
7. `NEW` creates the **New panel**. With this panel it is possible to start new simulator kernels on different computers or to link already existing kernels. The **New panel** is described in section 12.2.3.
8. `PUT` and `GET` change the net and the patterns within the selected kernel.
9. `SHOW INFO` selects the information, which is displayed in the `list window`.
10. `DONE` closes the **rpc panel**.

12.2.2 RPC Setup Panel

With the help of this panel the current settings of the marked simulator kernel are displayed and modified.

- `UDP` and `TCP` switch between the connection types UDP and TCP. If the connection type is changed, the current connection is terminated and a new connection is established. `UDP` and `TCP` are the two possible protocols that are build directly upon the internet protocol (IP). `UDP` (unreliable datagram protocol) is a little bit faster than `TCP`. Since `TCP` is more reliable we recomend it for usage with `xguirpc`.
- `NORMAL` and `COOP-MASTER` set the mode of the kernel. The function of the coop-master mode has not been implemented in this release of the SNNS.
- `Standard timeout` The waiting interval of the graphical user-interface, which is used to determine when an rpc error should be displayed.
- `Long duration timeout` The long duration timeout field is responsible for the waiting time in the learning process. It specifies the time `xguirpc` will wait for a `kernlrpc` program to finish a training cycle. If there is no response within that period of time `xguirpc` will assume an errror in the program execution. This value

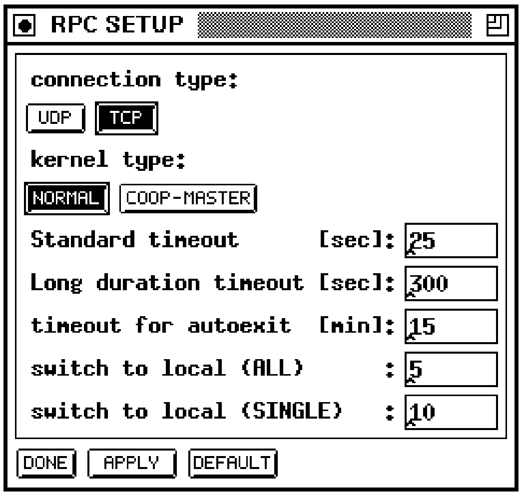

Figure 12.2: The RPC Setup Panel

should be rather large for big nets or pattern sets, otherwise an error will be displayed even if there is actually none.

- `Timeout for autoexit` is responsible to specify how long a simulator kernel will wait without receiving any messages before it automatically stops. This value is entered in minutes.
- `switch to local (ALL)` Specifies the threshold for asynchronous execution mode. If there are fewer training cycles specified in the control panel, `xguirpc` will wait for the remote host to finish learning (synchronous mode). If there are more training cycles specified, `xguirpc` will start training on the remote host and then immediately switch back to 'direct local' to allow the user further input as long as `kernlrpc` is executing training on the remote host (asynchronous mode). This value is relevant only for the training of all patterns.
- `switch to local (SINGLE)` Same as `switch to local (ALL)` but with relevance only to learning a single pattern.
- `APPLY` All entered values are set only after this button is pressed.
- `DEFAULT` will use the default values.
- `DONE` closes the window.

12.2.3 New Panel

The **New panel** is used to start new simulator kernels on different computers. A computer host name is required in the first line (it is also possible to select one from the list by clicking on it with the left mouse button). The second line specifies the path to the desired rpc kernel. Remember to seltect an rpc kernel of the correct computer architecture, if you are working with a heterogeneous workstation cluster. Different calling parameters can be added in the third line.

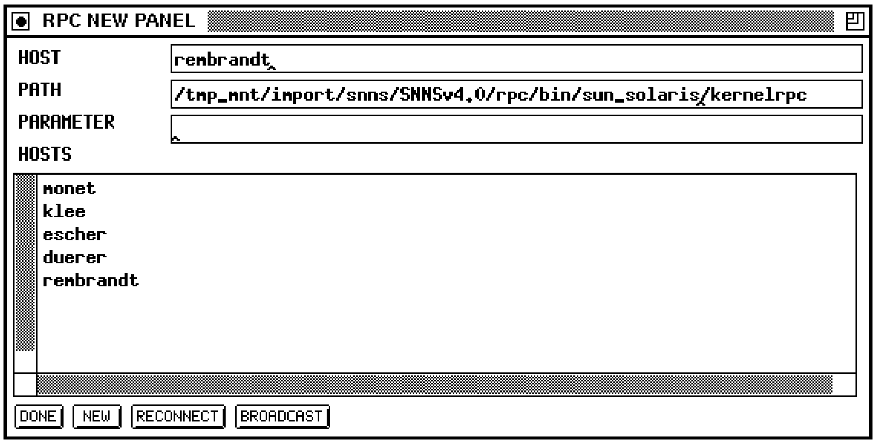

Figure 12.3: The RPC New Panel

The `NEW` button starts a new rpc kernel. To start the program, **`xguirpc`** uses the 'rsh'-instruction to execute a command on a remote computer system. The newly started simulator kernel receives the computer name, portmap number and version of the computer on which the graphical user-interface is running. After a successful start of the simulator kernel, the kernel registers with the graphical user-interface. The kernel is then included in the 'host-list' of the **rpc panel**.

If the new kernel is not listed in the 'host-list' of the **rpc panel** within a couple of seconds, it might be due to the fact that the given path is invalid on the specified machine (e.g. wrong mount prefix). In this case try to determine the correct path and try again.

With the `RECONNECT` and the `BROADCAST` buttons, different already existing simulator kernels can be connected with the graphical user-interface. The difference between those two buttons is that `RECONNECT` selects a specific computer and `BROADCAST` sends a signal to all computers in the LAN. It is not guaranteed that all computers will actually receive the signal.

12.2.4 Transfer Panels

To transfer patterns or networks to or from a local simulator kernel, the transfer panels can be used.

If the **Get** panel is chosen, it is necessary to have 'direct-local' as a current host and to have a remote simulator kernel marked with the mouse. The simulator kernel, which the nets and/or the patterns will be transfered to, can be chosen with the two arrows. Note that help files will be created in the directory /tmp as the transfer takes place. Those files will be deleted after the transfer. It is necessary to make sure that enough disk space in /tmp is available.

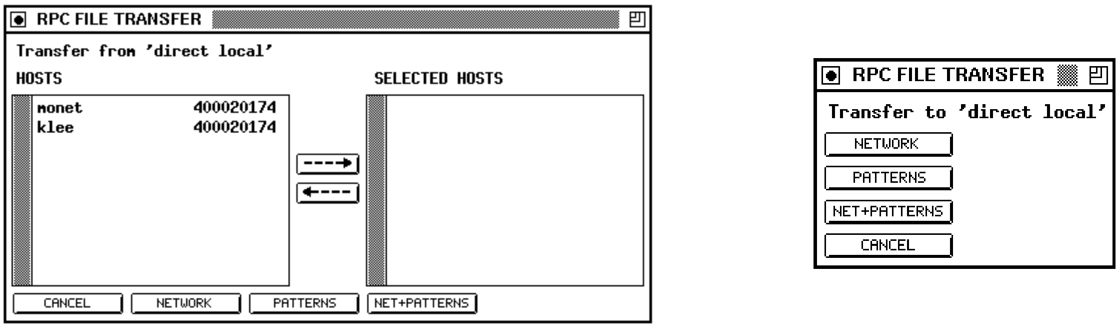

Figure 12.4: The RPC Transfer Panels. Put on the left, Get on the right.

12.2.5 Show Info Panel

With this panel it is possible to select the information, displayed in the **SNNS RPC** panel. Items can be added or deleted with the two arrows. The order of the selected items can be chosen with `UP IN LIST` or `DOWN IN LIST`.

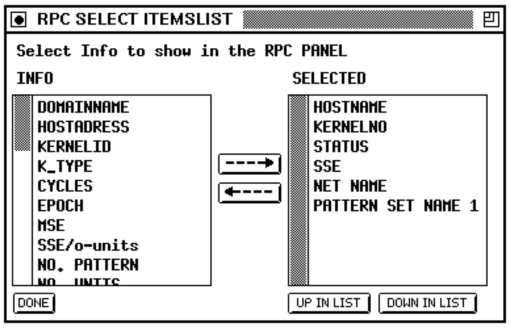

Figure 12.5: The RPC Info Panel

The possible fields are:

- `HOST-NAME`: the computer name on which the simulator kernel is running,
- `KERNEL-NAME`: the version number under which the simulator kernel is registered at the portmapper (this information can also be displayed with 'rpcinfo -p'),
- `STATUS`: the last known state of the simulator kernel,
- `DOMAINNAME`: the domain name of the network,
- `HOSTADDRESS`:the host address of the simulator kernel,
- `K-TYPE`: the type of the simulator kernel ("coop-master" and "normal" are possible, but coop-master is not implemented in the current version),
- `KERNEL_ID`: id of the simulator kernel in xgui,
- `CYCLES`: number of cycles the kernel used to train,

- `EPOCH`: number of epochs the kernel used to train,
- `SSE`: the sse error value,
- `MSE`: the mse error value,
- `SSE/o-units`: the sse error value divided bv the number of output units,
- `NO. PATTERN`: the number of patterns,
- `NO. UNITS`: the number of units,
- `NO. INPUT`: the number of input units,
- `NO. HIDDEN`: the number of hidden units,
- `NO. OUTPUT`: the number of output units,
- `NET NAME`: the network name,
- `PATTERN SET NAME`: the name of the pattern sets,
- `LEARN`: the selected learning parameters,
- `UPDATE`: the selected update parameters,
- `INIT`: the selected initialization parameters.

Note that all information is usually not displayed immediately, but with a small delay.

12.3 RPC Errors

If an rpc function is called, there may be additional errors besides regular SNNS errors, such as the call of the `DeletePattern` function without a defined pattern in the network. Because `rpcgen` is not generating error code within the rpc routines the result of these functions has to be checked for errors. Since these routines return a NULL value in case of an error, the return value can be checked for NULL and the system can then call an error routine, which determines the exact cause of the error. The rpc library provides such an error routine. Because the rpc mechanism can not determine if there is a net-adapter malfunction or if one of the cables is defect, the rpc mechanism only determines that there is no response from anyone within a given time interval. The error cause, delivered by the error function is not that an adapter card is defect, but that the time interval was exceeded (time-out error). This is the most common type of error and the communication module tries to detect the error with the help of the graphical user-interface.

To accomplish such a task, the communication module remembers if the last inquiry was performed synchronously or asynchronously. If the inquiry was asynchronous the probable cause is that the server is not ready for a new task, because it has to finish the current one. Therefore the simulator kernel reports to the graphical user-interface as soon as it has finished its task. This enables the communication module to update all status informations about the simulator kernel. The user will be informed about an error, and if available what kind of error, over a ‘pop up’ window.

Due to the separation of the graphical user-interface and the simulator kernel, the error tolerance has increased. If there is for example an error in the graphical user-interface, only the interface will be terminated and after a new start of the graphical user-interface and its linking to the simulator kernel, the user is able to proceed exactly where the error occured.

If an error occurs in the kernel the situation is less complicated. In that case simply a new net has to be started, since the communication module intercepts all maskable signals and saves the current net before it ends its operation correctly.

This kind of independence can be used to train multiple nets in parallel. All that has to be done is to load multiple nets with multiple simulator kernels and to start the training process asynchronously with the desired parameters. Now the graphical user-interface can be terminated, while the simulator kernel trains the nets. It is possible to interrupt the training and to evaluate the current state of the training process with the graphical user-interface. After that, the training can be resumed. In case the simulator kernel has already finished training the net it will wait a given period of time before it saves the net and ends its operations. In contrast to the batch version of SNNS, `snnsbat`, it is possible to select all actions of the kernel from a menu and to evaluate the current training, e.g. the current epochs and error rate.

14.12 Snns2c

Synopsis: `snns2c` <network> [<C-filename> [<function-name>]]

where:

`<network>`	is the name of the SNNS network file,
`<C-filename>`	is the name of the output file
`<function-name>`	is the name of the procedure in the application.

This tool compiles an SNNS network file into an executable C source. It reads a network file `<network.net>` and generates a C source named `<C-filename>`. The network can be called now as a function named `<function-name>`. If the parameter `<function-name>` is missing, the name of `<C-filename>` is taken without the ending "*.c". If this parameter is also missing, the name of the network file is chosen and fitted with a new ending for the output file. This name without ending is also used for the function name.

It is not possible to train the generated net, SNNS has to be used for this purpose. After completion of network training with SNNS, the tool SNNS2C is used to integrate the trained network as a C function into a separate application.

This program is also an example how to use the SNNS kernel interface for loading a net and changing the loaded net into another format. All data and all SNNS functions – except the activation functions – are placed in a single C function.

Note:
SNNS2C does not support sites. Any networks created with SNNS that make use of the site feature can not be converted to C source by this tool. Output functions are not supported, either.

The program can translate the following network-types:

- Feedforward networks trained with Backpropagation and all variants of it like Quickprop, RPROP etc.
- Radial Basis Functions
- Partially-recurrent Elman and Jordan networks
- Time Delay Neural Networks (TDNN)
- Dynamic Learning Vector Quantisation (DLVQ)
- Backpropagation Through Time (BPTT, QPTT, BBPTT)
- Counterpropagation Networks

While the use of SNNS or any parts of it in commercial applications requires a special agreement/licensing from the developers, the use of trained networks generated with SNNS2C is hereby granted without any fees for any purpose, provided proper academic credit to the SNNS team is given in the documentation of the application.

14.12.1 Program flow

Because the compilation of very large nets may require some time, the program outputs messages, indicating which state of compilation is passed at the moment.

loading net... the network file is loaded with the function offered by the kernel user interface.

dividing net into layers ... all units are grouped into layers where all units have the same type and the same activation function. There must not exist any dependencies between the units of the layers except the connections of SPECIAL HIDDEN units to themselves in Elman and Jordan networks or the links of the BPTT-networks.

sorting layers... these layers are sorted in topological order, e.g. first the input layer, then the hidden layers followed by the output layers and at last the special hidden layers. A layer which has sources in another layer of the same type is updated later as the source layer.

writing net... selects the needed activation functions and writes them to the C-source file. After that, the procedure for pattern propagation is written.

14.12.2 Including the Compiled Network in the Own Application

Interfaces: All generated networks may be called as C functions. This functions have the form:

`int`*function-name*`(float *in, float *out, int init)`

where `in` and `out` are pointers to the input and output arrays of the network. The init flag is needed by some network types and it's special meaning is explained in 14.12.3. The function normally returns the value 0 (OK). Other return values are explained in section 14.12.3.

The generated C-source can be compiled separately. To use the network it's necessary to include the generated header file (*.h) which is also written by SNNS2C. This header file contains a prototype of the generated function and a record which contains the number of input and output units also.

Example: If a trained network, was saved as "myNetwork.net" and compiled with

```
snns2c myNetwork.net
```

then the generated Network can be compiled with

```
gcc -c myNetwork.c
```

To include the network in your own application the header file must be included. There should also two arrays being provided, one for the input and one for the output of the network. The number of inputs and outputs can be derived from a record in the header file. This struct is named like the function which contains the compiled network and has the suffix REC to mark the record. So the number of input units is determined with myNetworkREC.NoOfInput and the number of outputs with myNetworkREC.NoOfOutput in this example. Hence, your own application should contain:

```
⋮

#include "myNetwork.h"

⋮

float *netInput, *netOutput; /* Input and Output arrays of the Network */
netInput  = malloc(myNetworkREC.NoOfInput  * sizeof(float));
netOutput = malloc(myNetworkREC.NoOfOutput * sizeof(float));

⋮

myNetwork(netInput, netOutput, 0)

⋮
```

Don't forget to link the object code of the network to your application

14.12.3 Special Network Architectures

Normally, the architecture of the network and the numbers of the units are kept. Therefore a dummy unit with the number 0 is inserted in the array which contains the units. Some architectures are translated with other special features.

TDNN: Generally, a layer in a time delay neural network consists of feature units and their delay units. SNNS2C generates code only containing the feature units. The delay units are only additional activations in the feature unit. This is possible because every delay unit has the same link weights to it's corresponding source units as its feature unit.

So the input layer consists only of its prototype units, too. Therefore it's not possible to present the whole input pattern to the network. This is not necessary because it can be presented step by step to the inputs. This is useful for a real-time application, with the newest feature units as inputs. To mark a new sequence the *init*-flag (parameter of the function) can be set to 1. After this, the delays are filled when the init flag is set to 0 again. To avoid meaningless outputs the function returns NOT_VALID until the delays are filled again.

There is a new variable in the record of the header file for TDNNs. It is called "MinDelay" and is the minimum number of time steps which are needed to get a valid output after the *init*-flag was set.

CPN: Counterpropagation doesn't need the output layer. The output is calculated as a weighted sum of the activations of the hidden units. Because only one hidden unit has the activation 1 and all others the activation 0, the output can be calculated with the winner unit, using the weights from this unit to the output.

DLVQ: Here no output units are needed either. The output is calculated as the bias of the winner unit.

BPTT: If all inputs are set to zero, the net is not initialized. This feature can be chosen by setting the *init*-flag.

14.12.4 Activation Functions

Supported Activation Functions: Following activation functions are implemented in SNNS2C

Supported Activation Functions	
Act_Logistic	Act_StepFunc
Act_Elliott	Act_Identity
Act_BSB	Act_IdentityPlusBias
Act_TanH	Act_RBF_Gaussian
Act_TanHPlusBias	Act_RBF_MultiQuadratic
Act_TanH_Xdiv2	Act_RBF_ThinPlateSpline
Act_Perceptron	Act_TD_Logistic
Act_Signum	Act_TD_Elliott
Act_Signum0	

Including Own Activation Functions: The file "tools/sources/functions.h" contains two arrays: One array with the function names (ACT_FUNC_NAMES) and one for the macros which represent the function (ACT_FUNCTIONS). These macros are realized as character strings so they can be written to the generated C-source.

The easiest way to include an own activation function is to write the two necessary entries in the first position of the arrays. After that the constant "ActRbfNumber" should be increased. If a new Radial Basis function should be included, the entries should be appended at the end without increasing ActRbfNumber. An empty string ("") should still be the last entry of the array Act_FUNC_NAMES because this is the flag for the end of the array.

14.12.5 Error messages

Here is the list of possible error messages and their brief description.

not enough memory: a dynamic memory allocation failed.

can't load file: SNNS network-file wasn't found.

can't open file: same as can't load or disk full.

wrong parameters: wrong kind or number of parameters.

net contains illegal cycles: there are several possibilities:

- a connection from a unit which is not a SPECIAL HIDDEN unit to itself.
- two layers are connected to each other when not exactly one of them is SPECIAL HIDDEN.
- cycles over more than two layers which don't match the architecture of Jordan networks.

BPTT-networks have no restrictions concerning links.

can't find the function *actfunc*: the activation function *actfunc* is not supported.

net is not a CounterPropagation network: Counterpropagation-networks need a special architecture: one input, one output and one hidden layer which are fully connected.

net is not a Time Delay Neural Network: The SNNS TDNNs have a very special architecture. They should be generated by the BIGNET-tool. In other cases there is no guaranty for successful compilation.

not supported network type: There are several network types which can't be compiled with the SNNS2C. The SNNS2C-tool will be maintained and so new network types will be implemented.

unspecified Error: should not occur, it's only a feature for user defined updates.

unknown error code : *number*: must not occur, please report to the SNNS-Team.

Bibliography

[Ama89] Shun-Ichi Amari. *Characteristics of Encoded Associative Memory.* Springer Verlag 1989, 1989.

[BD95] Michael R. Berthold and Jay Diamond. Boosting the performance of rbf networks with dynamic decay adjustment. In G. Tesauro, D. S. Touretzky, and T.K. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, 1995.

[CG87] G. A. Carpenter and S. Grossberg. Stable self-organization of pattern recognition codes for analog input patterns. *Applied Optics*, 26:4919–4930, 1987.

[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.

[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.

[Hud92] M.J. Hudak. Rce classifiers: Theory and practice. *Cybernetics and Systems*, pages 483–515, 1992.

[RCE82] D.L. Reilly, L.N. Cooper, and C. Elbaum. A neural model for category learning. *Biol. Cybernet.*, 45, 1982.

[Weh94] C. Wehrfritz. Neuronale netze als statistische methode zur erklärung von klassifikationen. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Statistik 1, 1994.