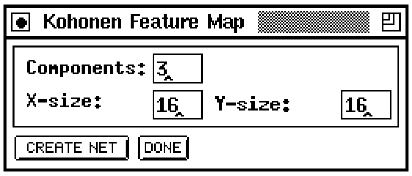

Figure 7.16: The BigNet window for the SOM architecture

- *X-size:* The width of the competitive layer. When learning is performed, the x-size value must be specified by the fifth learning parameter.
- *Y-size:* The length of the competitive layer. The number of hidden (competitive) units equals X-size $*$ Y-size.

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

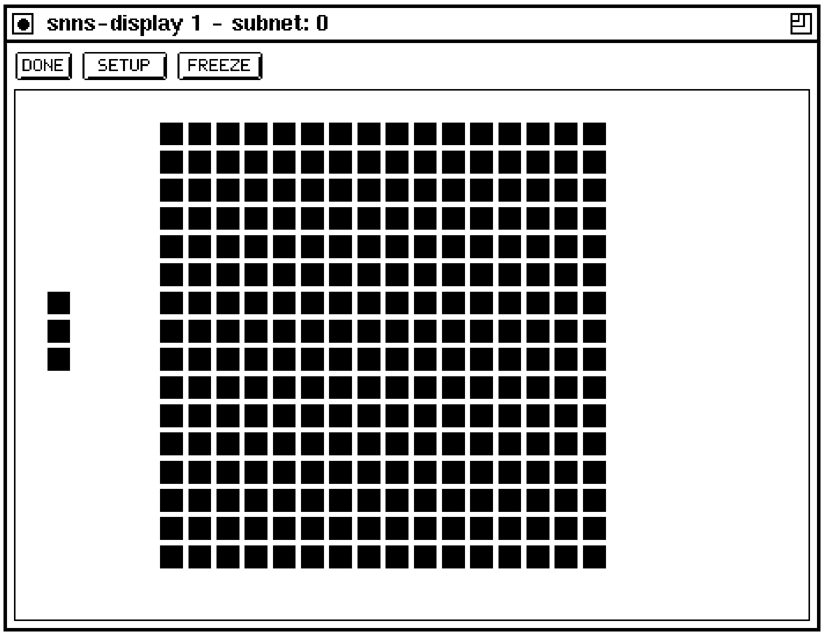

Figure 7.17: An Example SOM

7.5 BigNet for Autoassociative Memory Networks

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

Each unit of the world layer has a link to the corresponding unit in the learning layer. The learning layer is connected as a clique.

Figure 7.18: The BigNet Window for Autoassociative Memory Networks

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

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

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

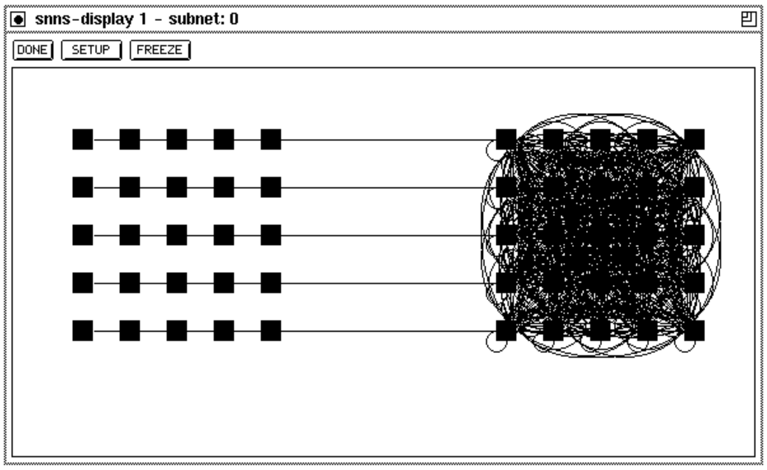

Figure 7.19: An Example Autoassociative Memory

7.6 BigNet for Partial Recurrent Networks

7.6.1 BigNet for Jordan Networks

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

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

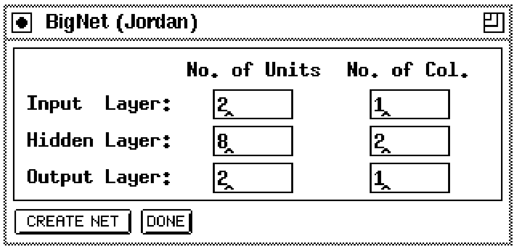

Figure 7.20: The BigNet Window for Jordan Networks

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

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

7.6.2 BigNet for Elman Networks

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

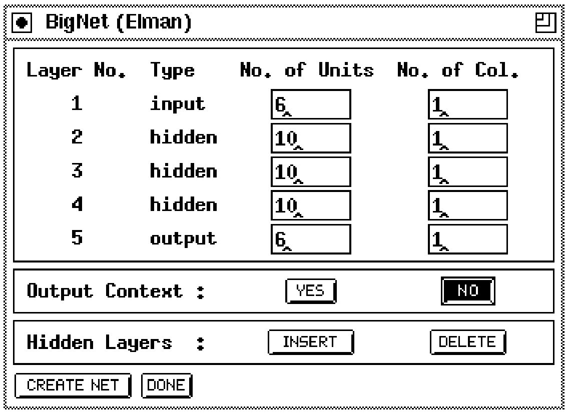

Figure 7.21: The BigNet Window for Elman Networks

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

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

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

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

Chapter 8

Network Analyzing Tools

8.1 Inversion

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

8.1.1 The Algorithm

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

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

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

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

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

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

and for the adaption value of the unit activation follows

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

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

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

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

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

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

8.1.2 Inversion Display

The inversion algorithm is called by clicking the $\boxed{\texttt{INVERSION}}$ button in the manager panel. Picture 8.1 shows an example of the generated display.

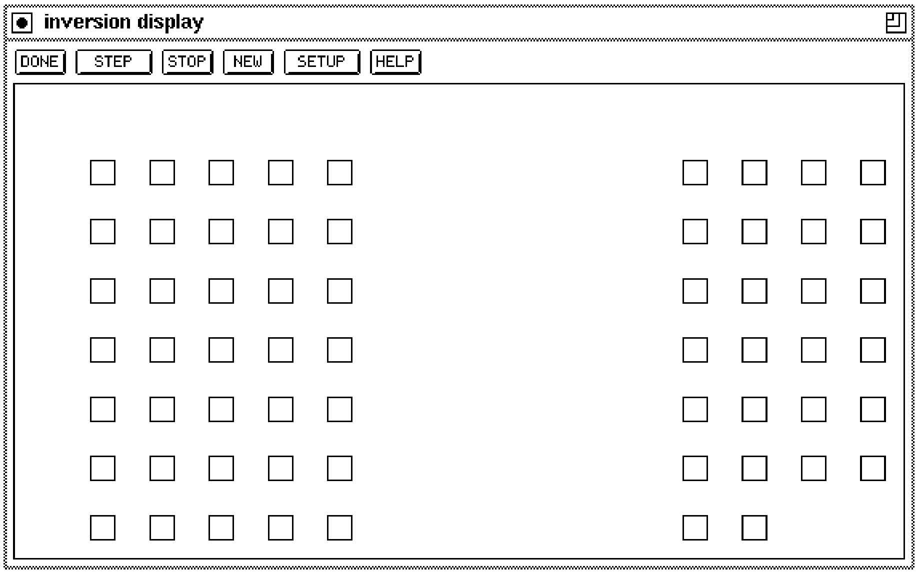

Figure 8.1: The Inversion Display

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

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

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

   where cycle is the number of the current iteration, inversion error is the sum of the squared error of the output units for the current input pattern, and error units are all units that have an activation that differs more than the value of δ_{max} from the target activation.
3. $\boxed{\texttt{STOP}}$: Interrupts the iteration. The status of the network remains unchanged. The interrupt causes the current activations of the units to be displayed on the screen. A click to the $\boxed{\text{STEP}}$ button continues the algorithm from its last state. Alternatively the algorithm can be reset before the restart by a click to the $\boxed{\text{NEW}}$ button, or continued with other parameters after a change in the setup. Since there is no automatic recognition of infinite loops in the implementation, the $\boxed{\text{STOP}}$ button is also necessary when the algorithm obviously does not converge.
4. $\boxed{\texttt{NEW}}$ Resets the network to a defined initial status. All variables are assigned the values in the setup panel. The iteration counter is set to zero.
5. $\boxed{\texttt{SETUP}}$: Opens a pop-up window to set all variables associated with the inversion. These variables are:

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

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

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

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

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

The network is displayed in the lower part of the window according to the settings of the last opened 2D–display window. Size, color, and orientation of the units are read from that display pointer.

8.1.3 Example Session

The inversion display may be called before or after the network has been trained. A pattern file for the network has to be loaded prior to calling the inversion. A target output of the network is defined by selecting one or more units in the 2D–display by clicking the middle mouse button. After setting the variables in the setup window, the inversion run is started by clicking the start button. At regular intervals, the inversion gives a status report on the shell window, where the progress of the algorithm can be observed. When there are no more error units, the program terminates and the calculated input pattern is displayed. If the algorithm does not converge, the run can be interrupted with the stop button and the variables may be changed. The calculated pattern can be tested for correctness by selecting all input units in the 2D–display and then deselecting them immediately again. This copies the activation of the units to the display. It can then be defined and tested with the usual buttons in the control panel. The user is advised to delete the generated pattern, since its use in subsequent learning cycles alters the behavior of the network which is generally not desirable.

Figure 8.2 shows an example of a generated input pattern (left). Here the minimum active units for recognition of the letter 'V' are given. The corresponding original pattern is shown on the right.

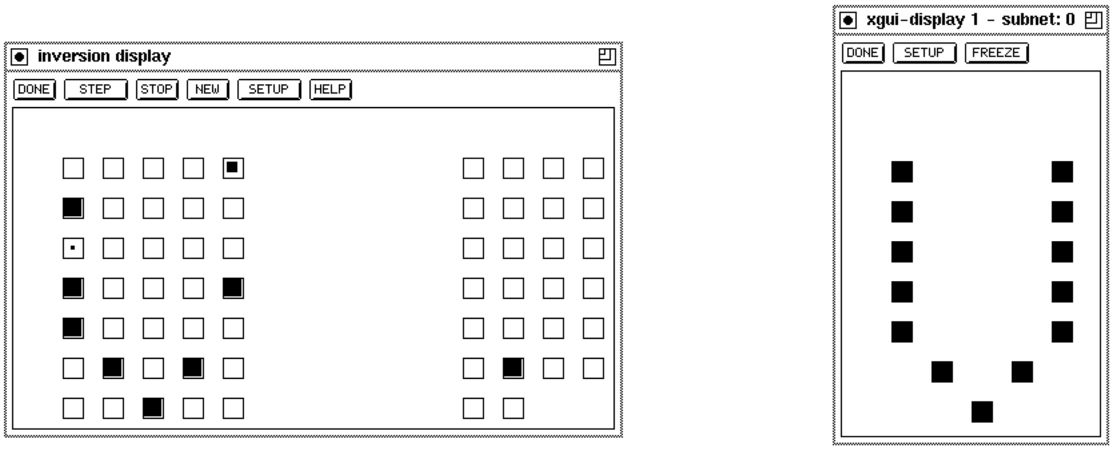

Figure 8.2: An Example of an Inversion Display (left) and the original pattern for the letter V

8.2 Network Analyzer

The network analyzer is a tool to visualize different types of graphs. An overview of these graphs is shown in table 8.1. This tool was especially developed for the prediction of time series with partial recurrent networks, but is also useful for regular feedforward networks. Its window is opened by selecting the entry `ANALYZER` in the menu under the `GUI` button.

The x-y graph is used to draw the activations or outputs of two units against each other. The t-y graph displays the activation (or output) of a unit during subsequent discrete time steps. The t-e graph makes it possible to visualize the error during subsequent discrete time steps.

Type	Axes
x–y	hor.: activation or output of a unit x ver.: activation or output of a unit y
t–y	hor.: time t ver.: activation or output of a unit y
t–e	hor.: time t ver.: error e

Table 8.1: The different types of graphs, which can be visualized with the network analyzer.

On the right side of the window, there are different buttons with the following functions:

`ON` : This button is used to "switch on" the network analyzer. If the Network Analyzer is switched on, every time a pattern has been propagated through the network, the network analyzer updates its display.

`LINE` : The points will be connected by a line, if this button is toggled.

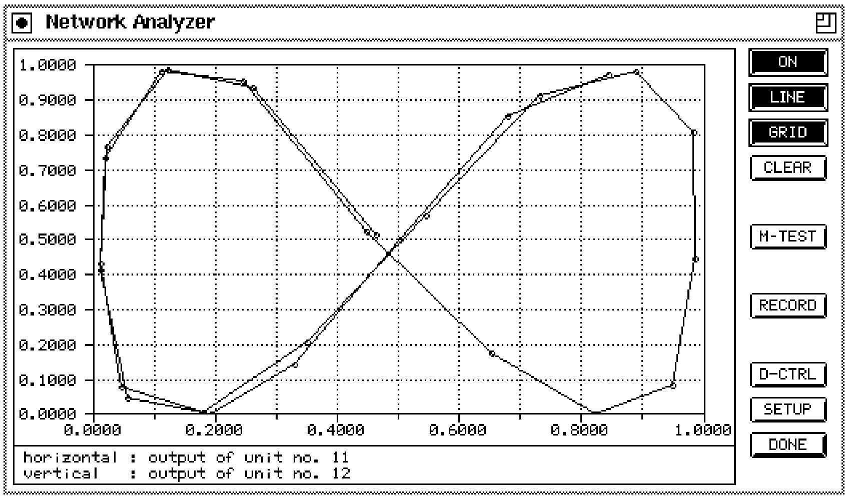

Figure 8.3: The Network Analyzer Window.

GRID : Displays a grid. The number of rows and columns of the grid can be specified in the network analyzer setup.

CLEAR : This button clears the graph in the display. The time counter will be reset to 1. If there is an active M--TEST operation, this operation will be killed.

M-TEST : A click on this button corresponds to several clicks on the TEST button in the control panel. The number n of TEST operations to be executed can be specified in the Network Analyzer setup. Once pressed, the button remains active until all n TEST operations have been executed or the M--TEST operation has been killed, e.g. by clicking the STOP button in the control panel.

RECORD : If this button activated, the points will not only be shown on the display, but their coordinates will also be saved in a file. The name of this file can be specified in the setup of the Network Analyzer.

D-CTRL : Opens the display control window of the Network Analyzer. The description of this window follows below.

SETUP : Opens the Network Analyzer setup window. The description of the setup follows in the next subsection.

DONE : Closes the network analyzer window. An active M--TEST operation will be killed.

8.2.1 The Network Analyzer Setup

The setup window can be opened by pressing the `SETUP` button on the right side of the network analyzer window. The shape of the setup window depends on the type of graph to display (see fig. 8.4).

The setup window consists of five parts (see fig. 8.4):

1. To select the type of graph (see table 8.1) to be displayed, press the corresponding button `X-Y` , `T-Y` or `T-E`.

2. The second part of the setup window is used to specify some attributes about the axes. The first line contains the values for the axes in horizontal direction, the second line these for the vertical axes. The columns `min` and `max` define the area to be displayed. The numbers of the units, whose activation or output values should be drawn have to be specified in the column `unit`. The last column `grid` the number of columns and rows of the grid can be varied. The labeling of the axes is dependent on these values, too.

3a. The selection between showing the activation or the output of a unit along the x– or y–axes can be made here. To draw the output of a unit click on `OUT` and to draw the activation of a unit click on `ACT`.

3b. Different types of error curves can be drawn:

$\sum_i \lvert t_i - o_i \rvert$	For each output unit the difference between the generated output and the teaching output is computed. The error is computed as the sum of the absolute values of the differences. If `AVE` is toggled, the result is divided by the number of output units, giving the average error per output unit.
$\sum_i \lvert t_i - o_i \rvert^2$	The error is computed as above, but the square of the differences is taken instead of the absolute values. With `AVE` the mean squared deviation is computed.
$\lvert t_j - o_j \rvert$	Here the deviation of only a single output unit is processed. The number of the unit is specified as `unit j`.

4. `m-test`: Specifies the number of `TEST` operations, which have to be executed when clicking on `M-TEST` button.

   `time`: Sets the time counter to the given value.

5. The name of the file, in which the visualized data can be saved by activating the `RECORD` button, can be specified here. The filename will be automatically extended by the suffix '.rec'. To change the filename, the `RECORD` button must not be activated.

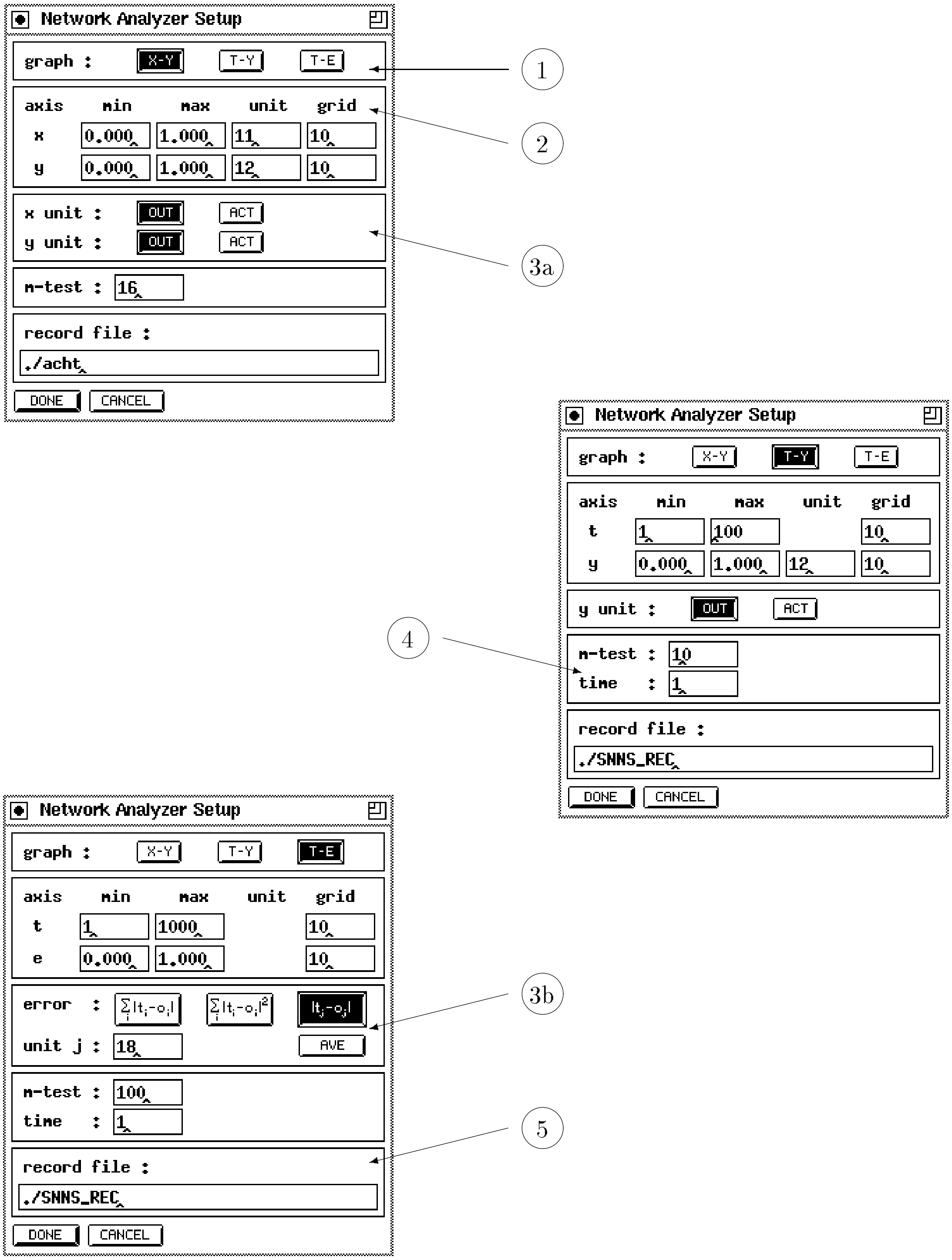

Figure 8.4: The Network Analyzer SetupWindows: the setup window for a x-y–graph (top), the setup window for a t-y–graph (middle) and the setup window for a t-e–graph (bottom).

When the setup is left by clicking on $\boxed{\texttt{CANCEL}}$ all the changes made in the setup are lost.

When leaving the setup by pressing the `DONE` button, the changes will be accepted if no errors could be detected.

8.2.2 The Display Control Window of the Network Analyzer

The display control window appears, when clicking on `D-CTRL` button on the right side of the network analyzer window. This windows is used to easily change the area in the display of the network analyzer.

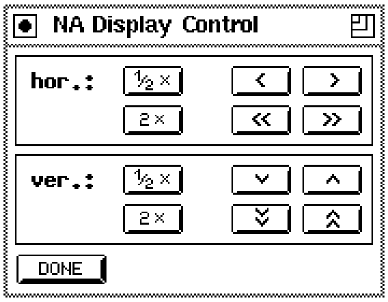

Figure 8.5: The display control window of the network analyzer

The buttons to change the range of the displayed area in horizontal direction have the following functions:

$\boxed{1/2\times}$	:	The length of the interval is to be bisected. The lower bound remains.
$\boxed{2\times}$	:	The length of the interval is to be doubled. The lower bound remains.
$\boxed{<}$	:	Shifts the range to the left by the width of a grid column.
$\boxed{>}$	:	Shifts the range to the right by the width of a grid column.
$\boxed{\ll}$	:	Shifts the range to the left by the length of the interval.
$\boxed{\gg}$	:	Shifts the range to the right by the length of the interval.

The buttons to change the range in vertical direction have a corresponding function. To close the display control window, press the `DONE` button.

Chapter 9

Neural Network Models and Functions

The following chapter introduces the models and learning functions implemented in SNNS. A strong emphasis is placed on the models that are less well known. They can not, however, be explained exhaustively here. We refer interested users to the literature.

9.1 Backpropagation Networks

9.1.1 Vanilla Backpropagation

The standard backpropagation learning algorithm introduced by [RM86] and described already in section 3.3 is implemented in SNNS. It is the most common learning algorithm. Its definition reads as follows:

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

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

9.1.2 Enhanced Backpropagation

An enhanced version of backpropagation uses a momentum term and flat spot elimination. It is listed among the SNNS learning functions as `BackpropMomentum`.

The momentum term introduces the old weight change as a parameter for the computation of the new weight change. This avoids oscillation problems common with the regular backpropagation algorithm when the error surface has a very narrow minimum area. The new weight change is computed by

$$\Delta w_{ij}(t+1) = \eta * \delta_j * o_i + \alpha \Delta w_{ij}(t)$$

α is a constant specifying the influence of the momentum.

The effect of these enhancements is that flat spots of the error surface are traversed relatively rapidly with a few big steps, while the step size is decreased as the surface gets rougher. This adaption of the step size increases learning speed significantly.

Note that the old weight change is lost every time the parameters are modified, new patterns are loaded, or the network is modified.

9.1.3 Batch Backpropagation

Batch backpropagation has a similar formula as vanilla backpropagation. The difference lies in the time when the update of the links takes place. While in vanilla backpropagation an update step is performed after each single pattern, in batch backpropagation all weight changes are summed over a full presentation of all training patterns (one epoch). Only then, an update with the accumulated weight changes is performed. This update behavior is especially well suited for training pattern parallel implementations where communication costs are critical.

9.1.4 Backpropagation with Weight Decay

Weight Decay was introduced by P. Werbos ([Wer88]). It decreases the weights of the links while training them with backpropagation. In addition to each update of a weight by backpropagation, the weight is decreased by a part d of its old value. The resulting formula is

$$\Delta w_{ij}(t+1) = \eta\, \delta_j\, o_i - d\, w_{ij}(t)$$

The effect is similar to the pruning algorithms (see chapter 10). Weights are driven to zero unless reinforced by backpropagation. For further information, see [Sch94].

9.2 Quickprop

One method to speed up the learning is to use information about the curvature of the error surface. This requires the computation of the second order derivatives of the error function. Quickprop assumes the error surface to be locally quadratic and attempts to jump in one step from the current position directly into the minimum of the parabola.

Quickprop [Fah88] computes the derivatives in the direction of each weight. After computing the first gradient with regular backpropagation, a direct step to the error minimum is attempted by

$$\Delta(t+1)w_{ij} = \frac{S(t+1)}{S(t) - S(t+1)} \Delta(t) w_{ij}$$

where:

w_{ij}	weight between units i and j
$\Delta(t+1)$	actual weight change
$S(t+1)$	partial derivative of the error function by w_{ij}
$S(t)$	the last partial derivative

9.3 RPROP

9.3.1 Changes in Release 3.3

The implementation of Rprop has been changed in two ways: First, the implementation now follows a slightly modified adaptation scheme. Essentially, the backtracking step is no longer performed, if a jump over a minimum occurred. Second, a weight-decay term is introduced. The weight-decay parameter α (the third learning parameter) determines the relationship of two goals, namely to reduce the output error (the standard goal) and to reduce the size of the weights (to improve generalization). The composite error function is:

$$E = \sum (t_i - o_i)^2 + 10^{-\alpha} \sum w_{ij}^2$$

Important: Please note that the weight decay parameter α denotes the exponent, to allow comfortable input of very small weight-decay. A choice of the third learning parameter $\alpha = 4$ corresponds to a ratio of weight decay term to output error of $1 : 10000 (1 : 10^4)$.

9.3.2 General Description

Rprop stands for '**R**esilient back**prop**agation' and is a local adaptive learning scheme, performing supervised batch learning in multi-layer perceptrons. For a detailed discussion see also [Rie93], [RB93]. The basic principle of Rprop is to eliminate the harmful influence of the size of the partial derivative on the weight step. As a consequence, only the sign of the derivative is considered to indicate the *direction* of the weight update. The *size* of the weight change is exclusively determined by a weight-specific, so-called 'update-value' $\triangle_{ij}^{(t)}$:

$$\triangle w_{ij}^{(t)} = \begin{cases} -\triangle_{ij}^{(t)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ +\triangle_{ij}^{(t)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\ 0 & , \text{ else} \end{cases} \tag{9.1}$$

where $\frac{\partial E}{\partial w_{ij}}^{(t)}$ denotes the summed gradient information over all patterns of the pattern set ('batch learning').

It should be noted, that by replacing the $\triangle_{ij}(t)$ by a constant update-value $\triangle$, equation (9.1) yields the so-called 'Manhattan'-update rule.

The second step of Rprop learning is to determine the new update-values $\triangle_{ij}(t)$. This is based on a sign-dependent adaptation process.

$$\triangle_{ij}^{(t)} = \begin{cases} \eta^+ * \triangle_{ij}^{(t-1)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ \eta^- * \triangle_{ij}^{(t-1)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\ \triangle_{ij}^{(t-1)} & , \text{ else} \end{cases} \tag{9.2}$$

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

In words, the adaptation-rule works as follows: Every time the partial derivative of the corresponding weight w_{ij} changes its sign, which indicates that the last update was too big and the algorithm has jumped over a local minimum, the update-value $\triangle_{ij}^{(t)}$ is decreased by the factor η^-. If the derivative retains its sign, the update-value is slightly increased in order to accelerate convergence in shallow regions. Additionally, in case of a change in sign, there should be no adaptation in the succeeding learning step. In practice, this can be achieved by setting $\frac{\partial E}{\partial w_{ij}}^{(t-1)} := 0$ in the above adaptation rule (see also the description of the algorithm in the following section).

In order to reduce the number of freely adjustable parameters, often leading to a tedious search in parameter space, the increase and decrease factor are set to fixed values ($\eta^- := 0.5, \eta^+ = 1.2$).

For Rprop tries to adapt its learning process to the topology of the error function, it follows the principle of 'batch learning' or 'learning by epoch'. That means, that weight-update and adaptation are performed after the gradient information of the whole pattern set is computed.

9.3.3 Parameters

The Rprop algorithm takes three parameters: the initial update-value $\triangle_0$, a limit for the maximum step size, $\triangle_{max}$, and the weight-decay exponent α (see above).

When learning starts, all update-values are set to an initial value $\triangle_0$. Since $\triangle_0$ directly determines the size of the first weight step, it should be chosen according to the initial values of the weights themselves, for example $\triangle_0 = 0.1$ (default setting). The choice of this value is rather uncritical, for it is adapted as learning proceeds.

In order to prevent the weights from becoming too large, the maximum weight-step determined by the size of the update-value, is limited. The upper bound is set by the second parameter of Rprop, $\triangle_{max}$. The default upper bound is set somewhat arbitrarily to $\triangle_{max} = 50.0$. Usually, convergence is rather insensitive to this parameter as well. Nevertheless, for some problems it can be advantageous to allow only very cautious (namely small) steps, in order to prevent the algorithm getting stuck too quickly in suboptimal local minima. The minimum step size is constantly fixed to $\triangle_{min} = 1e^{-6}$.

9.4 Backpercolation

Backpercolation 1 (Perc1) is a learning algorithm for feedforward networks. Here the weights are not changed according to the error of the output layer as in backpropagation, but according to a unit error that is computed separately for each unit. This effectively reduces the amount of training cycles needed.

The algorithm consists of five steps:

1. A pattern is propagated through the network and the global error is computed.
2. The gradient δ is computed and propagated back through the hidden layers as in backpropagation.
3. The error ϵ in the activation of each hidden neuron is computed. This error specifies the value by which the output of this neuron has to change in order to minimize the global error Err.
4. All weight parameters are changed according to ϵ.
5. If necessary, an adaptation of the error magnifying parameter λ is performed once every learning epoch.

The third step is divided into two phases: First each neuron receives a message $\Delta\mu$, specifying the proposed change in the activation of the neuron (message creation - MCR). Then each neuron combines the incoming messages to an optimal compromise, the internal error ϵ of the neuron (message optimization - MOP). The MCR phase is performed in forward direction (from input to output), the MOP phase backwards.

The internal error ϵ_k of the output units is defined as $\epsilon_k = \lambda(d_k - \phi_k)$, where λ is the global error magnification parameter.

Unlike backpropagation Perc1 does not have a learning parameter. Instead it has an error magnification parameter λ. This parameter may be adapted after each epoch, if the total mean error of the network falls below the threshold value θ.

When using backpercolation with a network in SNNS the initialization function `Random_-Weights_Perc` and the activation function `Act_TanH_Xdiv2` should be used.

9.5 Counterpropagation

9.5.1 Fundamentals

Counterpropagation was originally proposed as a pattern-lookup system that takes advantage of the parallel architecture of neural networks. Counterpropagation is useful in pattern mapping and pattern completion applications and can also serve as a sort of bidirectional associative memory.

When presented with a pattern, the network classifies that pattern by using a learned reference vector. The hidden units play a key role in this process, since the hidden layer performs a competitive classification to group the patterns. Counterpropagation works best on tightly clustered patterns in distinct groups.

Two types of layers are used: The hidden layer is a Kohonen layer with competitive units that do unsupervised learning; the output layer is a Grossberg layer, which is fully connected with the hidden layer and is not competitive.

When trained, the network works as follows. After presentation of a pattern in the input layer, the units in the hidden layer sum their inputs according to

$$\text{net}_j = \sum_i w_{ij} o_i$$

and then compete to respond to that input pattern. The unit with the highest net input wins and its activation is set to 1 while all others are set to 0. After the competition, the output layer does a weighted sum on the outputs of the hidden layer.

$$a_k = \text{net}_k = \sum_j w_{jk} o_j$$

Let c be the index of the winning hidden layer neuron. Since o_c is the only nonzero element in the sum, which in turn is equal to one, this can be reduced to

$$a_k = w_{ck}$$

Thus the winning hidden unit activates a pattern in the output layer.

During training, the weights are adapted as follows:

1. A winner of the competition is chosen in response to an input pattern.
2. The weights between the input layer and the winner are adjusted according to

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

   All the other weights remain unchanged.
3. The output of the network is computed and compared to the target pattern.
4. The weights between the winner and the output layer are updated according to

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

   All the other weights remain unchanged.

9.5.2 Initializing Counterpropagation

For Counterpropagation networks three initialization functions are available: CPN_Rand_-Pat, CPN_Weights_v3.2, and CPN_Weights_v3.3. See section 4.6 for a detailed description of these functions.

Note:
In SNNS versions 3.2 and 3.3 there was only the initialization function `CPN_Weights` available. Although it had the same name, there was a significant difference between the two.

The older version, still available now as `CPN_Weights_v3.2` selected its values from the hyper**cube** defined by the two initialization parameters. This resulted in an uneven distribution of these values after they had been normalized, thereby biasing the network towards a certain (unknown) direction. The newer version, still available now as `CPN_Weights_v3.3` selected its values from the hyper**sphere** defined by the two initialization parameters. This resulted in an even distribution of these values after they had been normalized. However it had the disadvantage of having an exponential time complexity, thereby making it useless for networks with more than about 15 input units. The influence of the parameters on these two functions is given below.

Two parameters are used which represent the minimum (a) and maximum (b) of the range out of which initial values for the second (Grossberg) layer are selected at random. The vector w_i of weights leading to unit i of the Kohonen layer are initialized as normalized vectors (length 1) drawn at random from part of a hyper-sphere (hyper-cube). Here, min and max determine which part of the hyper body is used according to table 9.1.

min (a)	max (b)	vectors out of ...
$a \geq 0$	$b \geq 0$	positive sector
$a \geq 0$	$b < 0$	whole hyper-sphere
$a < 0$	$b \geq 0$	whole hyper-sphere
$a < 0$	$b < 0$	negative sector

Table 9.1: Influence of minimum and maximum on the initialization of weight vectors for CPN and SOM.

9.5.3 Counterpropagation Implementation in SNNS

To use Counterpropagation in SNNS the following functions and variables have to be selected.

One of the above mentioned initialization functions, the update function `CPN_Order`, and the learning function `Counterpropagation`. The activation function of the units may be set to any of the sigmoidal functions available in SNNS.

9.6 Dynamic Learning Vector Quantization (DLVQ)

9.6.1 DLVQ Fundamentals

The idea of this algorithm is to find a natural grouping in a set of data ([SK92], [DH73]. Every data vector is associated with a point in a d-dimensional data space. The hope is that the vectors $\vec{x}$ of the same class form a cloud or a cluster in data space. The algorithm presupposes that the vectors $\vec{x}$ belonging to the same class w_i are distributed normally with a mean vector $\vec{\mu_i}$, and that all input vectors are normalized. To classify a feature vector $\vec{x}$ measure the Euclidian distance $||\vec{\mu} - \vec{x}||^2$ from $\vec{x}$ to all other mean vectors $\vec{\mu}$ and assign $\vec{x}$ to the class of the nearest mean. But what happens if a pattern $\vec{x_A}$ of class w_A is assigned to a wrong class w_B? Then for this wrong classified pattern the two mean vectors $\vec{\mu_A}$ and $\vec{\mu_B}$ are moved or trained in the following way:

- The reference vector $\vec{\mu_A}$ which the wrong classified pattern belongs to, and which is the nearest neighbor to this pattern, is moved a little bit towards this pattern.
- The mean vector $\vec{\mu_B}$, to which a pattern of class w_A is assigned wrongly, is moved away from it.

The vectors are moved using the rule:

$$w_{ij} = w_{ij} + \eta(o_i - w_{ij}).$$

where w_{ij} is the weight[1] between the output o_i of a input unit i and a output unit j. η is the learning parameter. By choosing it less or greater than zero, the direction of movement of a vector can be influenced.

The DLVQ algorithm works in the following way:

1. Load the (normalized) training data, and calculate for every class the mean vector μ. Initialize the net with these vectors. This means: Generate a unit for every class and initialize its weights with the corresponding values.
2. Now try to associate every pattern in the training set with a reference vector. If a trainings vector $\vec{x}$ of a class w_A is assigned to a class w_B then do the following:
   (a) Move the vector $\vec{\mu_A}$ which is nearest to $\vec{x_A}$ in its direction.
   (b) Move the mean vector $\vec{\mu_B}$, to which $\vec{x_A}$ is falsely assigned to away from it.

   Repeat this procedure until the number of correctly classified vectors no longer increases.
3. Now calculate, from the vectors of a class w_A associated with a wrong class w_B, a new prototype vector μ_A. For every class, choose one of the new mean vectors and add it to the net. Return to step 2.

9.6.2 DLVQ in SNNS

To start DLVQ, the learning function `DLVQ`, the update function `DLVQ_Update`, and the init function `DLVQ_Weights` have to be selected in the corresponding menus. The init functions of `DLVQ` differ a little from the normal function: if a DLVQ net is initialized, all hidden units are deleted.

As with learning rules CC and RCC the text field `CYCLE` in the control panel does **not** specify the number of learning cycles. This field is used to specify the maximal number of class units to be generated for each class during learning. The number of learning cycles is entered as the third parameter in the control panel (see below).

[1] Every mean vector $\vec{\mu}$ of a class is represented by a class unit. The elements of these vectors are stored in the weights between class unit and the input units.

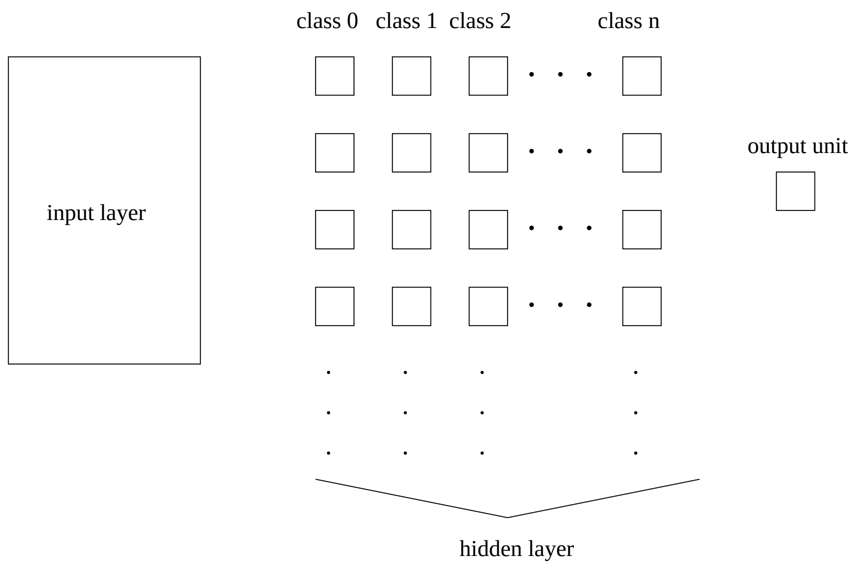

Figure 9.1: Topology of a net which was trained with DLVQ.

1. η^+: learning rate, specifies the step width of the mean vector $\vec{\mu_A}$, which is nearest to a pattern $\vec{x_A}$, towards this pattern. Remember that $\vec{\mu_A}$ is moved only, if $\vec{x_A}$ is not assigned to the correct class w_A. A typical value is 0.03.
2. η^-: learning rate, specifies the step width of a mean vector $\vec{\mu_B}$, to which a pattern of class w_A is falsely assigned to, away from this pattern. A typical value is 0.03. Best results can be achieved, if the condition $\eta^+ = \eta^-$ is satisfied.
3. Number of cycles you want to train the net before additive mean vectors are calculated.

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

The output layer must consist of only one unit. At the start of the learning phase it does not matter whether the output layer and the input layer are connected. If hidden units exist, they are fully connected with the input layer. The links between these layers contain the values of the the mean vectors. The output layer and the hidden layer are fully connected. All these links have the value 1 assigned.

The output pattern contains information on which class the input pattern belongs to. The lowest class must have the name 0. If there are n classes, the n-th class has the name $n-1$. If these conditions are violated, an error occurs. Figure 9.1 shows the topology of a net. In the bias of every class unit its class name is stored. It can be retrieved by clicking on a class unit with right mouse button.

Note: In the first implementation of DLVQ the input patterns were automatically normalized by the algorithm. This step was eliminated, since is produced undesired behavior in some cases. Now the user has to take all necessary steps to normalize the input vectors correctly before loading them into SNNS.

9.6.3 Remarks

This algorithm was developed in the course of a masters thesis without knowledge of the original LVQ learning rules ([KKLT92]). Only later we found out that we had developed a new LVQ algorithm: It starts with the smallest possible number of hidden layers and adds new hidden units only when needed. Since the algorithm generates the hidden layer dynamically during the learning phase, it was called dynamic LVQ (DLVQ).

It is obvious that the algorithm works only if the patterns belonging to the same class have some similarities. Therefore the algorithm best fits classification problems such as recognition of patterns, digits, and so on. This algorithm succeeded in learning 10000 digits with a resolution of 16×16 pixels. Overall the algorithm generated 49 hidden units during learning.

9.7 Backpropagation Through Time (BPTT)

This is a learning algorithm for recurrent networks that are updated in discrete time steps (non-fixpoint networks). These networks may contain any number of feedback loops in their connectivity graph. The only restriction in this implementation is that there may be no connections between input units[2]. The gradients of the weights in the recurrent network are approximated using an feedforward network with a fixed number of layers. Each layer t contains all activations $a_i(t)$ of the recurrent network at time step t. The highest layer contains the most recent activations at time $t = 0$. These activations are calculated synchronously, using only the activations at $t = 1$ in the layer below. The weight matrices between successive layers are all identical. To calculate an exact gradient for an input pattern sequence of length T, the feedforward network needs $T + 1$ layers if an output pattern should be generated after the last pattern of the input sequence. This transformation of a recurrent network into a equivalent feedforward network was first described in [MP69], p. 145 and the application of backpropagation learning to these networks was introduced in [RHW86].

To avoid deep networks for long sequences, it is possible to use only a fixed number of layers to store the activations back in time. This method of *truncated backpropagation through time* is described in [Zip90] and is used here. An improved feature in this implementation is the combination with the quickprop algorithm by [Fah88] for weight adaption. The number of additional copies of network activations is controlled by the parameter `backstep`. Since the setting of `backstep` virtually generates a hierarchical network with `backstep` +1 layers and error information during backpropagation is diminished very rapidly in deep networks, the number of additional activation copies is limited by `backstep` ≤ 10.

There are three versions of backpropagation through time available:

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

[2]This case may be transformed into a network with an additional hidden unit for each input unit and a single connection with unity weight from each input unit to its corresponding hidden unit.

layers and the weights are adapted using the formula for backpropagation with momentum term after each pattern. The momentum term uses the weight change during the previous pattern. Using small learning rates `eta`, `BPTT` is especially useful to start adaption with a large number of patterns since the weights are updated much more frequently than in batch-update.

BBPTT: Batch backpropagation through time.
The gradient for each weight is calculated for each pattern as in `BPTT` and then averaged over the whole training set. The momentum term uses update information closer to the true gradient than in `BPTT`.

QPTT: Quickprop through time.
The gradient in quickprop through time is calculated as in `BBPTT`, but the weights are adapted using the substantially more efficient quickprop-update rule.

A recurrent network has to start processing a sequence of patterns with defined activations. All activities in the network may be set to zero by applying an input pattern containing only zero values. If such all-zero patterns are part of normal input patterns, an extra input unit has to be added for reset control. If this reset unit is set to 1, the network is in the free running mode. If the reset unit *and* all normal input units are set to 0, all activations in the network are set to 0 and all stored activations are cleared as well.

The processing of an input pattern $I(t)$ with a set of non-input activations $a_i(t)$ is performed as follows:

1. The input pattern $I(t)$ is copied to the input units to become a subset of the existing unit activations $a_i(t)$ of the whole net.
2. If $I(t)$ contains only zero activations, all activations $a_i(t+1)$ and all stored activations $a_i(t), a_i(t-1), \ldots, a_i(t-$ `backstep`$)$ are set to 0.0.
3. All activations $a_i(t+1)$ are calculated synchronously using the activation function and activation values $a_i(t)$.
4. During learning, an output pattern $O(t)$ is always compared with the output subset of the *new* activations $a_i(t+1)$.

Therefore there is exactly *one* synchronous update step between an input and an output pattern with the same pattern number.

If an input pattern has to be processed with more than one network update, there has to be a delay between corresponding input and output patterns. If an output pattern o^P is the n-th pattern after an input pattern i^P, the input pattern has been processed in $n+1$ update steps by the network. These $n+1$ steps may correspond to n hidden layers processing the pattern or a recurrent processing path through the network with $n+1$ steps. Because of this pipelined processing of a pattern sequence, the number of hidden layers that may develop during training in a fully recurrent network is influenced by the delay between corresponding input and output patterns. If the network has a defined hierarchical topology without shortcut connections between n different hidden layers, an output pattern should be the n-th pattern after its corresponding input pattern in the pattern file.

An example illustrating this relation is given with the delayed XOR network in the network file `xor-rec.net` and the pattern files `xor-rec1.pat` and `xor-rec2.pat`. With the patterns `xor-rec1.pat`, the task is to compute the XOR function of the previous input pattern. In `xor-rec2.pat`, there is a delay of 2 patterns for the result of the XOR of the input pattern. Using a fixed network topology with shortcut connections, the BPTT learning algorithm develops solutions with a different number of processing steps using the shortcut connections from the first hidden layer to the output layer to solve the task in `xor-rec1.pat`. To map the patterns in `xor-rec2.pat` the result is first calculated in the second hidden layer and copied from there to the output layer during the next update step[3].

The update function `BPTT-Order` performs the synchronous update of the network and detects reset patterns. If a network is tested using the TEST button in the control panel, the internal activations and the output activation of the output units are first overwritten with the values in the target pattern, depending on the setting of the button SHOW. To provide correct activations on feedback connections leading out of the output units in the following network update, all output activations are copied to the units initial activation values `i_act` after each network update and are copied back from `i_act` to `out` before each update. The non-input activation values may therefore be influenced before a network update by changing the initial activation values `i_act`.

If the network has to be reset by stepping over a reset pattern with the TEST button, keep in mind that after clicking TEST, the pattern number is increased first, the new input pattern is copied into the input layer second, and then the update function is called. So to reset the network, the current pattern must be set to the pattern directly preceding the reset pattern.

9.8 The Cascade Correlation Algorithms

Two cascade correlation algorithms have been implemented in SNNS, Cascade-Correlation and recurrent Cascade-Correlation. Both learning algorithms have been developed by Scott Fahlman ([FL91], [HF91], [Fah91]). Strictly speaking the cascade architecture represents a kind of meta algorithm, in which usual learning algorithms like Backprop, Quickprop or Rprop are embedded. Cascade-Correlation is characterized as a constructive learning rule. It starts with a minimal network, consisting only of an input and an output layer. Minimizing the overall error of a net, it adds step by step new hidden units to the hidden layer.

Cascade-Correlation is a supervised learning architecture which builds a near minimal multi-layer network topology. The two advantages of this architecture are that there is no need for a user to worry about the topology of the network, and that Cascade-Correlation learns much faster than the usual learning algorithms.

[3] If only an upper bound n for the number of processing steps is known, the input patterns may consist of windows containing the current input pattern together with a sequence of the previous $n-1$ input patterns. The network then develops a focus to the sequence element in the input window corresponding to the best number of processing steps.

9.8.1 Cascade-Correlation (CC)

9.8.1.1 The Algorithm

Cascade-Correlation (CC) combines two ideas: The first is the cascade architecture, in which hidden units are added only one at a time and do not change after they have been added. The second is the learning algorithm, which creates and installs the new hidden units. For each new hidden unit, the algorithm tries to maximize the magnitude of the correlation between the new unit's output and the residual error signal of the net.

The algorithm is realized in the following way:

1. CC starts with a minimal network consisting only of an input and an output layer. Both layers are fully connected.
2. Train all the connections ending at an output unit with a usual learning algorithm until the error of the net no longer decreases.
3. Generate the so-called candidate units. Every candidate unit is connected with all input units and with all existing hidden units. Between the pool of candidate units and the output units there are no weights.
4. Try to maximize the correlation between the activation of the candidate units and the residual error of the net by training all the links leading to a candidate unit. Learning takes place with an ordinary learning algorithm. The training is stopped when the correlation scores no longer improves.
5. Choose the candidate unit with the maximum correlation, freeze its incoming weights and add it to the net. To change the candidate unit into a hidden unit, generate links between the selected unit and all the output units. Since the weights leading to the new hidden unit are frozen, a new permanent feature detector is obtained. Loop back to step 2.

This algorithm is repeated until the overall error of the net falls below a given value. Figure 9.2 shows a net after 3 hidden units have been added.

9.8.1.2 Mathematical Background

The training of the output units tries to minimize the sum-squared error E:

$$E = \sum_p \frac{1}{2} \sum_o (y_{po} - t_{po})^2$$

where t_{po} is the desired and y_{po} is the observed output of the output unit o for a pattern p. The error E is minimized by gradient decent using

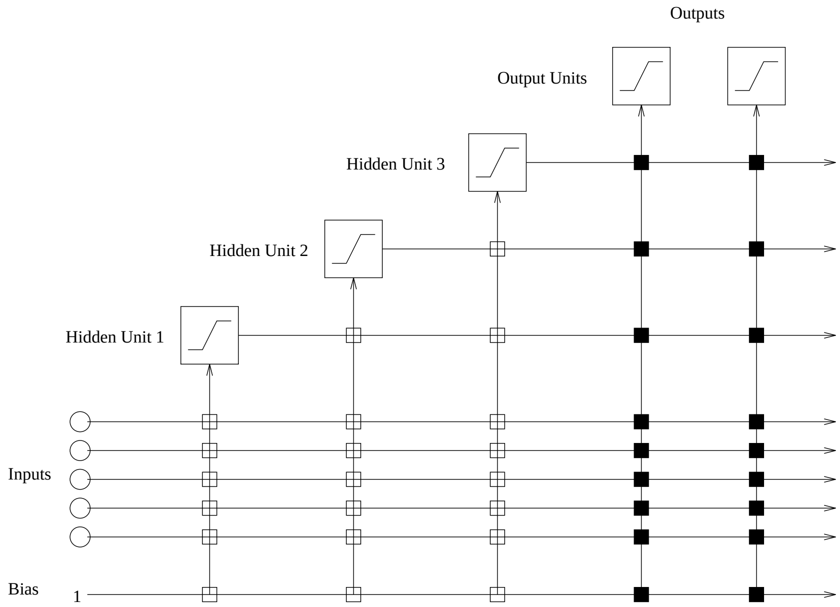

Figure 9.2: A neural net trained with cascade-correlation after 3 hidden units have been added. The vertical lines add all incoming activations. Connections with white boxes are frozen. The black connections are trained repeatedly.

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

where f'_p is the derivative of an activation function of a output unit o and I_{ip} is the value of an input unit or a hidden unit i for a pattern p. w_{io} denominates the connection between an input or hidden unit i and an output unit o.

After the training phase the candidate units are adapted, so that the correlation C between the value y_{po} of a candidate unit and the residual error e_{po} of an output unit becomes maximal. The correlation is given by Fahlman with:

$$
\begin{aligned}
C &= \sum_o \left| \sum_p (y_{po} - \bar{y_o})(e_{po} - \bar{e_o}) \right| \\
&= \sum_o \left| \sum_p y_{po} e_{po} - \bar{e_o} \sum_p y_{po} \right| \\
&= \sum_o \left| \sum_p y_{po} (e_{po} - \bar{e_o}) \right|,
\end{aligned}
$$

where $\bar{y_o}$ is the average activation of a candidate unit and $\bar{e_o}$ is the average error of an output unit over all patterns p. The maximization of C proceeds by gradient ascent using

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

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

9.8.2 Pruned-Cascade-Correlation (PCC)

9.8.2.1 The Algorithm

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

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

1. *Train the connections to the output layer*
2. Compute the selection criterion
3. *Train the candidates*
4. *Install the new hidden neuron*
5. Compute the selection criterion
6. Set each weight of the last inserted unit to zero and compute the selection criterion; if there exists a weight, whose removal would decrease the selection criterion, remove the link, which decreases the selection criterion **most**. Goto step 5 until a further removal would increase the selection criterion.
7. Compute the selection criterion; if it is greater than the one, computed before inserting the new hidden unit, notify the user that the net is getting too big.

9.8.2.2 Mathematical Background

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

9.8.3 Recurrent Cascade-Correlation (RCC)

9.8.3.1 The Algorithm

Recurrent Cascade-Correlation (RCC) is a recurrent version of Cascade-Correlation and can be used to train recurrent neural nets ([Elm90]).

Recurrent nets have some features that distinguish them from normal neural networks. For example they can be used to represent time implicitly rather than explicitly. One of the most commonly known architectures of recurrent neural nets is the Elman model, which assumes that the network operates in discrete time steps. The outputs of the network's hidden units at a time t are fed back for use as additional network inputs at time $t+1$. To store the output of the hidden units Elman introduced *context units*, which represent a kind of short-term memory (see fig. 9.3). To integrate the Elman model into the cascade architecture some changes are necessary: The hidden units' values are no longer fed back to all other hidden units. Instead every hidden unit has only one self recurrent link as shown in figure 9.4. This self recurrent link is trained along with the candidate unit's other input weights to maximize the correlation. When the candidate unit is added to the active network as hidden unit, the recurrent link is frozen along with all other links.

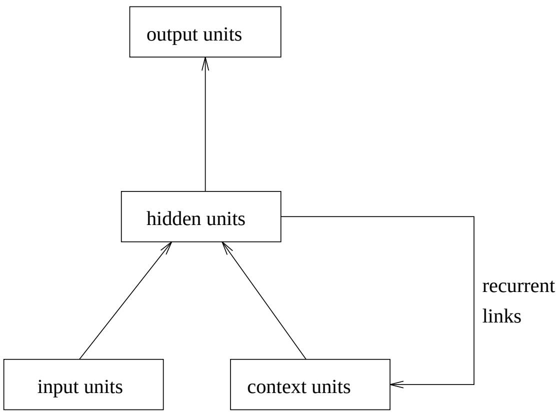

Figure 9.3: The Elman architecture of a recurrent neural net

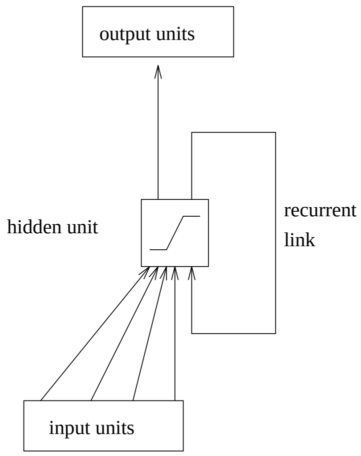

Figure 9.4: RCC architecture of a recurrent neural net.

9.8.3.2 Mathematical Background

The output of a recurrent unit r is given by:

$$V(t) \quad = \quad f_{act}\left(\sum_i I_i(t) w_{ir} + V(t-1) w_s\right),$$

where:

f_{act}: any activation function
w_{ir}: the weight of the connection between the input unit i and the unit r
w_s: the value of the self recurrent connection.
I_i: output of the input unit i

The derivatives of $V(t)$ with respect to w_{ir} and w_s are computed as follows:

$$\frac{\partial V}{\partial w_{ir}} = f'_{act}(t)(I_i(t) + w_s \frac{\partial V(t-1)}{\partial w_{ir}})$$

and

$$\frac{\partial V}{\partial w_s} = f'_{act}(t)(V(t-1) + w_s \frac{\partial V(t-1)}{\partial w_s}).$$

At $t = 0$, it is assumed that the unit's previous value and previous derivatives are all zero.

9.8.4 Using the Cascade Algorithms in SNNS

Networks that make use of the cascade correlation architecture can be created in SNNS in the same way as all other network types. The control of the training phase, however, is moved from the control panel to the special cascade window described below. The control panel is still used to specify the learning parameters, while the text field `CYCLE` does **not** specify as usual the number of learning cycles. This field is used here to specify the maximal number of hidden units to be generated during the learning phase. The number of learning cycles is entered in the cascade window. The learning parameters for the embedded learning functions Quickprop, Rprop and Backprop are described in chapter 4.4.

If the topology of a net is specified correctly, the program will automatically order the units and layers from left to right in the following way: input layer, hidden layer, output layer, and a candidate layer. [4] The hidden layer is generated with 5 units always having the same x-coordinate (i.e. above each other on the display).

The cascade correlation control panel and the cascade window (see fig. 9.5), is opened by clicking the `Cascade` button in the manager panel. The cascade window is needed to set the parameters of the learning algorithms CC and RCC. To start Cascade Correlation, learning function `CC`, update function `CC_Order` and init function `CC_Weights` in the corresponding menus have to be selected. Recurrent Cascade-Correlation is started in the same way, only that this time the functions `RCC`, `RCC_Update` and `RCC_Weights` have to be selected. If one of these functions is left out, a confirmer window with an error message pops up and learning does not start. The init functions of cascade differ from the normal init functions: upon initialization of a cascade net all hidden units are deleted.

The cascade window has the following text fields, buttons and menus:

[4] The candidate units are realized as special units in SNNS.

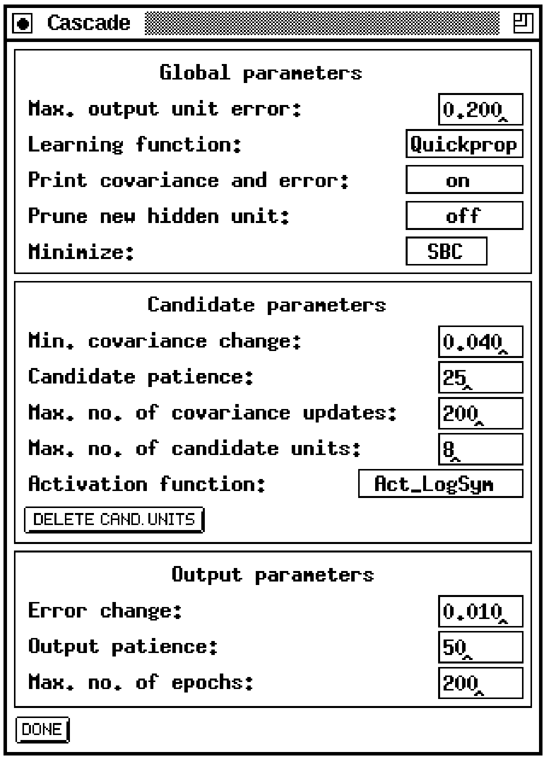

Figure 9.5: The cascade window

- `Global parameters:`
  - `Max. output unit error:`
    This value is used as abort condition for the learning algorithms CC and RCC. If the error of every single output unit is smaller than the given value learning will be terminated.
  - `Learning function:`
    Here, the learning function used to maximize the covariance or to minimize the net error can be selected from a pull down menu. Available learning functions are: `Quickprop`, `Rprop` and `Backprop`
  - `Print covariance and error:`
    If this menu item shows `on`, the development of the error and and the covariance of every candidate unit is printed. `off` prevents all outputs of the net.
  - `Prune new hidden unit:`
    This enables “Pruned-Cascade-Correlation”. It defaults to `off`, which means do not remove any weights from the new inserted hidden unit.
  - `Minimize:`
    The selection criterion according to which PCC tries to minimize. The default selection criterion is the “Schwarz’s Bayesian criterion”, other criteria available are “Akaikes information criterion” and the “conservative mean square error of

prediction". This option is ignored, unless PCC is enabled.

- `Candidate Parameters:`
  - `Min. covariance change:`
    The covariance must change by at least this fraction of its old value to count as a significant change. If this fraction is not reached, learning is halted and the candidate unit with the maximum covariance is changed into a hidden unit.
  - `Candidate patience:`
    After this number of steps the program tests whether there is a significant change of the covariance. The change is said to be significant if it is larger than the fraction given by `Min. covariance change`.
  - `Max. no. of covariance updates:`
    The maximum number of steps to calculate the covariance. After reaching this number, the candidate unit with the maximum covariance is changed to a hidden unit.
  - `Max. no. of candidate units:`
    The maximum number of candidate units trained at once.
  - `Activation function:`
    This menu item makes it possible to choose between different activation functions for the candidate units. The functions are: `Act_Logistic Act_LogSym, Act_Tanh, Act_Identity` and `Random`. `Random` is not a real activation function. It randomly assigns one of the other activation functions to each candidate unit. The function `Act_LogSym` is identical to `Act_Logistic`, except that it is shifted by -0.5 along the y-axis.
- `Output Parameters:`
  - `Error change:`
    analogous to `Min. covariance change`
  - `Output patience:`
    analogous to `Candidate patience`
  - `Max. no. of epochs:`
    analogous to `Max. no. of covariance updates`

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

9.9 Time Delay Networks (TDNNs)

9.9.1 TDNN Fundamentals

Time delay networks (or TDNN for short), introduced by Alex Waibel ([WHH$^+$89]), are a group of neural networks that have a special topology. They are used for position independent recognition of features within a larger pattern. A special convention for naming different parts of the network is used here (see figure 9.6)

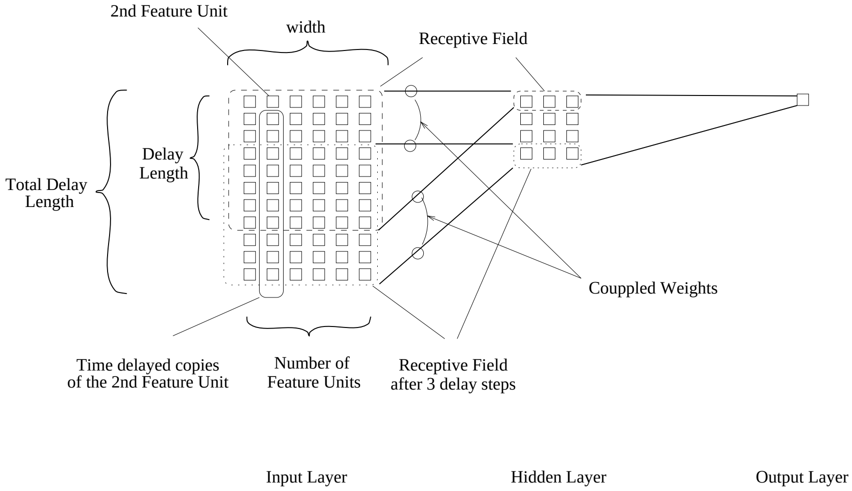

Figure 9.6: The naming conventions of TDNNs

- **Feature:** A component of the pattern to be learned.
- **Feature Unit:** The unit connected with the feature to be learned. There are as many feature units in the input layer of a TDNN as there are features.
- **Delay:** In order to be able to recognize patterns place or time-invariant, older activation and connection values of the feature units have to be stored. This is performed by making a copy of the feature units with all their outgoing connections in each time step, before updating the original units. The total number of time steps saved by this procedure is called *delay*.
- **Receptive Field:** The feature units and their delays are fully connected to the original units of the subsequent layer. These units are called *receptive field*. The receptive field is usually, but not necessarily, as wide as the number of feature units; the feature units might also be split up between several receptive fields. Receptive fields may overlap in the source plane, but do have to cover all feature units.
- **Total Delay Length:** The length of the layer. It equals the sum of the length of all delays of the network layers topological following the current one minus the number of these subsequent layers.
- **Coupled Links:** Each link in a receptive field is reduplicated for every subsequent step of time up to the total delay length. During the learning phase, these links are treated as a single one and are changed according to the average of the changes they would experience if treated separately. Also the units' bias which realizes a special sort of link weight is duplicated over all delay steps of a current feature unit. In figure 9.6 only two pairs of coupled links are depicted (out of 54 quadruples) for simplicity reasons.

The activation of a unit is normally computed by passing the weighted sum of its inputs to an activation function, usually a threshold or sigmoid function. For TDNNs this behavior is modified through the introduction of delays. Now all the inputs of a unit are each multiplied by the N delay steps defined for this layer. So a hidden unit in figure 9.6 would get 6 undelayed input links from the six feature units, and 7x6 = 48 input links from the seven delay steps of the 6 feature units for a total of 54 input connections. Note, that all units in the hidden layer have 54 input links, but only those hidden units activated at time 0 (at the top most row of the layer) have connections to the actual feature units. All other hidden units have the same connection pattern, but shifted to the bottom (i.e. to a later point in time) according to their position in the layer (i.e. delay position in time). By building a whole network of time delay layers, the TDNN can relate inputs in different points in time or input space.

Training in this kind of network is performed by a procedure similar to backpropagation, that takes the special semantics of coupled links into account. To enable the network to achieve the desired behavior, a sequence of patterns has to be presented to the input layer with the feature shifted within the patterns. Remember that since each of the feature units is duplicated for each frame shift in time, the whole history of activations is available at once. But since the shifted copies of the units are mere duplicates looking for the same event, weights of the corresponding connections between the time shifted copies have to be treated as one. First, a regular forward pass of backpropagation is performed, and the error in the output layer is computed. Then the error derivatives are computed and propagated backward. This yields different correction values for corresponding connections. Now all correction values for corresponding links are averaged and the weights are updated with this value.

This update algorithm forces the network to train on time/position independent detection of sub-patterns. This important feature of TDNNs makes them independent from error-prone preprocessing algorithms for time alignment. The drawback is, of course, a rather long, computationally intensive, learning phase.

9.9.2 TDNN Implementation in SNNS

The original time delay algorithm was slightly modified for implementation in SNNS, since it requires either variable network sizes or fixed length input patterns. Time delay networks in SNNS are allowed no delay in the output layer. This has the following consequences:

- The input layer has fixed size.
- Not the whole pattern is present at the input layer at once. Therefore one pass through the network is not enough to compute all necessary weight changes. This makes learning more computationally intensive.

The coupled links are implemented as one physical (i.e. normal) link and a set of logical links associated with it. Only the physical links are displayed in the graphical user interface. The bias of all delay units has no effect. Instead, the bias of the corresponding feature unit is used during propagation and backpropagation.

Activation Function

For time delay networks the new activation function `Act_TD_Logistic` has been implemented. It is similar to the regular logistic activation function `Act_Logistic` but takes care of the special coupled links. The mathematical notation is again

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

where o_i includes now also the predecessor units along logical links.

Update Function

The update function `TimeDelay_Order` is used to propagate patterns through a time delay network. It's behavior is analogous to the `Topological_Order` function with recognition of logical links.

Learning Function

The learning function `TimeDelayBackprop` implements the modified backpropagation algorithm discussed above. It uses the same learning parameters as standard backpropagation.

9.9.3 Building and Using a Time Delay Network

In SNNS, TDNNs should be generated only with the tool `BIGNET (Time Delay)`. This program automatically defines the necessary variables and link structures of TDNNs. The logical links are not depicted in the displays and can not be modified with the graphical editor. Any modifications of the units after the creation of the network may result in undesired behavior or even system failure!

After the creation of the net, the unit activation function `Act_TD_Logistic`, the update function `TimeDelay_Order`, and the learning function `TimeDelayBackprop` have to be assigned in the usual way.

NOTE: Only after the special time delay learning function has been assigned, will a save of the network also save the special logical links! A network saved beforehand will lack these links and be useless after a later load operation. Also using the TEST and STEP button will destroy the special time delay information unless the right update function (`TimeDelay_Order`) has been chosen.

Patterns must fit the input layer. If the application requires variable pattern length, a tool to segment these patterns into fitting pieces has to be applied. Patterns may also be generated with the graphical user interface. In this case, it is the responsibility of the user to supply enough patterns with time shifted features for the same teaching output to allow a successful training.

9.10 Radial Basis Functions (RBFs)

The following section describes the use of generalized radial basis functions inside SNNS. First, a brief introduction to the mathematical background of radial basis functions is given. Second, the special procedures of initialization and training of neural nets based on radial basis functions are described. At the end of the chapter a set of necessary actions to use radial basis functions with a specific application are given.

9.10.1 RBF Fundamentals

The principle of radial basis functions derives from the theory of functional approximation. Given N pairs $(\vec{x_i}, y_i)$ $(\vec{x} \in \Re^n, y \in \Re)$ we are looking for a function f of the form:

$$f(\vec{x}) = \sum_{i=1}^{K} c_i h(|\vec{x} - \vec{t_i}|)$$

h is the radial basis function and $\vec{t_i}$ are the K centers which have to be selected. The coefficients c_i are also unknown at the moment and have to be computed. $\vec{x_i}$ and $\vec{t_i}$ are elements of an n–dimensional vector space.

h is applied to the Euclidian distance between each center $\vec{t_i}$ and the given argument $\vec{x}$. Usually a function h which has its maximum at a distance of zero is used, most often the Gaussian function. In this case, values of $\vec{x}$ which are equal to a center $\vec{t}$ yield an output value of 1.0 for the function h, while the output becomes almost zero for larger distances.

The function f should be an approximation of the N given pairs $(\vec{x_i}, y_i)$ and should therefore minimize the following error function H:

$$H[f] = \sum_{i=1}^{N} (y_i - f(\vec{x_i}))^2 + \lambda \|Pf\|^2$$

The first part of the definition of H (the sum) is the condition which minimizes the total error of the approximation, i.e. which constrains f to approximate the N given points. The second part of H ($\|Pf\|^2$) is a stabilizer which forces f to become as smooth as possible. The factor λ determines the influence of the stabilizer.

Under certain conditions it is possible to show that a set of coefficients c_i can be calculated so that H becomes minimal. This calculation depends on the centers $\vec{t_i}$ which have to be chosen beforehand.

Introducing the following vectors and matrices $\vec{c} = (c_1, \cdots, c_K)^T, \vec{y} = (y_1, \cdots, y_N)^T$

$$G = \begin{pmatrix} h(|\vec{x_1} - \vec{t_1}|) & \cdots & h(|\vec{x_1} - \vec{t_K}|) \\ \vdots & \ddots & \vdots \\ h(|\vec{x_N} - \vec{t_1}|) & \cdots & h(|\vec{x_N} - \vec{t_K}|) \end{pmatrix}, \quad G_{\Box} = \begin{pmatrix} h(|\vec{t_1} - \vec{t_1}|) & \cdots & h(|\vec{t_1} - \vec{t_K}|) \\ \vdots & \ddots & \vdots \\ h(|\vec{t_K} - \vec{t_1}|) & \cdots & h(|\vec{t_K} - \vec{t_K}|) \end{pmatrix}$$

the set of unknown parameters c_i can be calculated by the formula:

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

By setting λ to 0 this formula becomes identical to the computation of the Moore Penrose inverse matrix, which gives the best solution of an under-determined system of linear equations. In this case, the linear system is exactly the one which follows directly from the conditions of an exact interpolation of the given problem:

$$f(\vec{x_j}) = \sum_{i=1}^{K} c_i h(|\vec{x_j} - \vec{t_i}|) \stackrel{!}{=} y_j \quad , \quad j = 1, \ldots N$$

The method of radial basis functions can easily be represented by a three layer feedforward neural network. The input layer consists of n units which represent the elements of the vector $\vec{x}$. The K components of the sum in the definition of f are represented by the units of the hidden layer. The links between input and hidden layer contain the elements of the vectors $\vec{t_i}$. The hidden units compute the Euclidian distance between the input pattern and the vector which is represented by the links leading to this unit. The activation of the hidden units is computed by applying the Euclidian distance to the function h. Figure 9.7 shows the architecture of the special form of hidden units.

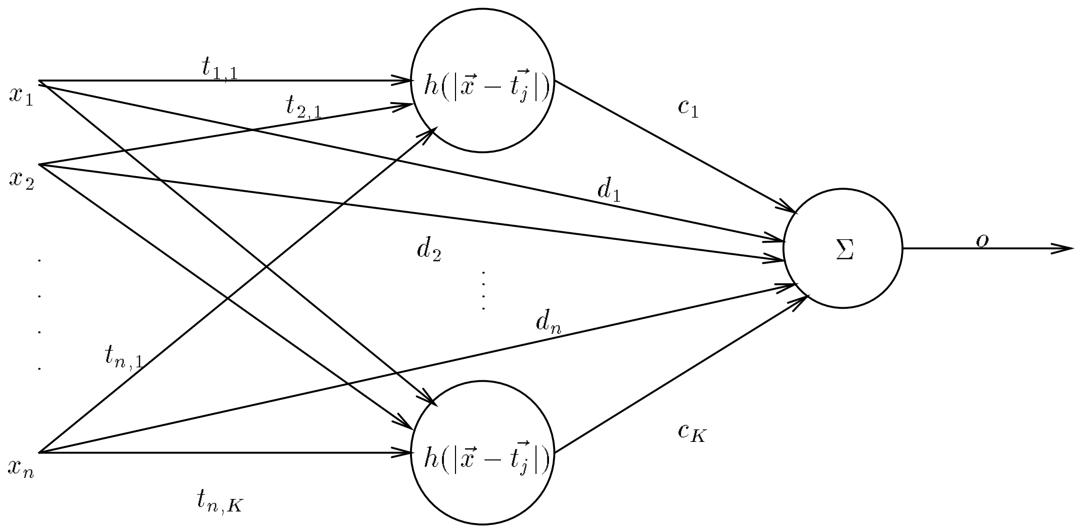

Figure 9.7: The special radial basis unit

The single output neuron gets its input from all hidden neurons. The links leading to the output neuron hold the coefficients c_i. The activation of the output neuron is determined by the weighted sum of its inputs.

The previously described architecture of a neural net, which realizes an approximation using radial basis functions, can easily be expanded with some useful features: More than one output neuron is possible which allows the approximation of several functions f around the same set of centers $\vec{t_i}$. The activation of the output units can be calculated by using a nonlinear invertible function σ (e.g. sigmoid). The bias of the output neurons and a direct connection between input and hidden layer (shortcut connections) can be used to improve the approximation quality. The bias of the hidden units can be used to modify the characteristics of the function h. All in all a neural network is able to represent the following set of approximations:

$$o_k(\vec{x}) = \sigma\left(\sum_{j=1}^{K} c_{j,k} h\left(|\vec{x} - \vec{t_j}|, p_j\right) + \sum_{i=1}^{n} d_{i,k} x_i + b_k\right) = \sigma\left(f_k(\vec{x})\right)\ , \quad k = 1, \ldots, m$$

This formula describes the behavior of a fully connected feedforward net with n input, K hidden and m output neurons. $o_k(\vec{x})$ is the activation of output neuron k on the input $\vec{x} = x_1, x_2, \ldots, x_n$ to the input units. The coefficients $c_{j,k}$ represent the links between hidden and output layer. The shortcut connections from input to output are realized by $d_{i,k}$. b_k is the bias of the output units and p_j is the bias of the hidden neurons which determines the exact characteristics of the function h. The activation function of the output neurons is represented by σ.

The big advantage of the method of radial basis functions is the possibility of a direct computation of the coefficients $c_{j,k}$ (i.e. the links between hidden and output layer) and the bias b_k. This computation requires a suitable choice of centers $\vec{t_j}$ (i.e. the links between input and hidden layer). Because of the lack of knowledge about the quality of the $\vec{t_j}$, it is recommended to append some cycles of network training after the direct computation of the weights. Since the weights of the links leading from the input to the output layer can also not be computed directly, there must be a special training procedure for neural networks that uses radial basis functions.

The implemented training procedure tries to minimize the error E by using gradient descent. It is recommended to use different learning rates for different groups of trainable parameters. The following set of formulas contains all information needed by the training procedure:

$$E = \sum_{k=1}^{m}\left(\sum_{i=1}^{N}(y_{i,k} - o_k(\vec{x_i}))^2\right) \quad , \quad \Delta\vec{t_j} = -\eta_1 \frac{\partial E}{\partial \vec{t_j}} \quad , \quad \Delta p_j = -\eta_2 \frac{\partial E}{\partial p_j}$$

$$\Delta c_{j,k} = -\eta_3 \frac{\partial E}{\partial c_{j,k}} \quad , \quad \Delta d_{i,k} = -\eta_3 \frac{\partial E}{\partial d_{i,k}} \quad , \quad \Delta b_k = -\eta_3 \frac{\partial E}{\partial b_k}$$

It is often helpful to use a momentum term. This term increases the learning rate in smooth error planes and decreases it in rough error planes. The next formula describes the effect of a momentum term on the training of a general parameter g depending on the additional parameter μ. Δg_{t+1} is the change of g during the time step $t+1$ while Δg_t is the change during time step t:

$$\Delta g_{t+1} = -\eta \frac{\partial E}{\partial g} + \mu \Delta g_t$$

Another useful improvement of the training procedure is the definition of a maximum allowed error inside the output neurons. This prevents the network from getting over-trained, since errors that are smaller than the predefined value are treated as zero. This in turn prevents the corresponding links from being changed.

9.10.2 RBF Implementation in SNNS

9.10.2.1 Activation Functions

For the use of radial basis functions, three different activation functions h have been implemented. For computational efficiency the square of the distance $r^2 = |\vec{x} - \vec{t}|^2$ is uniformly used as argument for h. Also, an additional argument p has been defined which represents the bias of the hidden units. The vectors $\vec{x}$ and $\vec{t}$ result from the activation and weights of links leading to the corresponding unit. The following radial basis functions have been implemented:

1. `Act_RBF_Gaussian` — the Gaussian function

$$h(r^2, p) = h(q, p) = e^{-pq} \quad \text{where} \quad q = |\vec{x} - \vec{t}|^2$$

2. `Act_RBF_MultiQuadratic` — the multiquadratic function

$$h(r^2, p) = h(q, p) = \sqrt{p + q} \quad \text{where} \quad q = |\vec{x} - \vec{t}|^2$$

3. `Act_RBF_ThinPlateSpline` – the *thin plate splines* function

$$\begin{aligned} h(r^2, p) = h(q, p) &= p^2 q \ln(p\sqrt{q}) \quad \text{where} \quad q = |\vec{x} - \vec{t}|^2 \\ &= (pr)^2 \ln(pr) \quad \text{where} \quad r = |\vec{x} - \vec{t}| \end{aligned}$$

During the construction of three layered neural networks based on radial basis functions, it is important to use the three activation functions mentioned above only for neurons inside the hidden layer. There is also only one hidden layer allowed.

For the output layer two other activation functions are to be used:

1. `Act_IdentityPlusBias`
2. `Act_Logistic`

`Act_IdentityPlusBias` activates the corresponding unit with the weighted sum of all incoming activations and adds the bias of the unit. `Act_Logistic` applies the sigmoid logistic function to the weighted sum which is computed like in `Act_IdentityPlusBias`.

In general, it is necessary to use an activation function which pays attention to the bias of the unit.

The last two activation functions converge towards infinity, the first converges towards zero. However, all three functions are useful as base functions. The mathematical preconditions for their use are fulfilled by all three functions and their use is backed by practical experience. All three functions have been implemented as base functions into SNNS.

The most frequently used base function is the Gaussian function. For large distances r, the Gaussian function becomes almost 0. Therefore, the behavior of the net is easy to predict if the input patterns differ strongly from all teaching patterns. Another advantage of the Gaussian function is, that the network is able to produce useful results without the use of shortcut connections between input and output layer.

9.10.2.2 Initialization Functions

The goal in initializing a radial basis function network is the optimal computation of link weights between hidden and output layer. Here the problem arises that the centers $\vec{t_j}$ (i.e. link weights between input and hidden layer) as well as the parameter p (i.e. the bias of the hidden units) must be set properly. Therefore, three different initialization procedures have been implemented which perform different tasks:

1. `RBF_Weights`: This procedure first selects evenly distributed centers $\vec{t_j}$ from the loaded training patterns and assigns them to the links between input and hidden layer. Subsequently the bias of all neurons (parameter p) inside the hidden layer is set to a value determined by the user and finally the links between hidden and output layer are computed.
2. `RBF_Weights_Redo`: In contrast to the preceding procedure only the links between hidden and output layer are computed. All other links and bias remain unchanged.
3. `RBF_Weights_Kohonen`: Using the self–organizing method of Kohonen feature maps, appropriate centers are generated on base of the teaching patterns. The computed centers are copied into the corresponding links. No other links and bias are changed.

It is necessary that valid patterns are loaded into SNNS to use the initialization. If no patterns are present upon starting any of the three procedures an alert box will occur showing the error. A detailed description of the procedures and the parameters used is given in the following paragraphs.

RBF_Weights Of the named three procedures `RBF_Weights` is the most comprehensive one. Here all necessary initialization tasks (setting link weights and bias) for a fully connected three layer feedforward network (without shortcut connections) can be performed in one single step. Hence, the choice of centers (i.e. the link weights between input and hidden layer) is rather simple: The centers are evenly selected from the loaded teaching patterns and assigned to the links of the hidden neurons. The selection process assigns the first teaching pattern to the first hidden unit, and the last pattern to the last hidden unit. The remaining hidden units receive centers which are evenly picked from the set of teaching patterns. If, for example, 13 teaching patterns are loaded and the hidden layer consists of 5 neurons, then the patterns with numbers 1, 4, 7, 10 and 13 are selected as centers.

Before a selected teaching pattern is distributed among the corresponding link weights it can be modified slightly with a random number. For this purpose, an initialization parameter (*deviation*, parameter 5) is set, which determines the maximum percentage of deviation allowed to occur randomly. To calculate the deviation, an inverse tangent function is used to approximate a normal distribution so that small deviations are more probable than large deviations. Setting the parameter *deviation* to 1.0 results in a maximum deviation of 100%. The centers are copied unchanged into the link weights if the deviation is set to 0.

A small modification of the centers is recommended for the following reasons: First, the number of hidden units may exceed the number of teaching patterns. In this case it is necessary to break the symmetry which would result without modification. This symmetry would render the calculation of the Moore Penrose inverse matrix impossible. The second reason is that there may be a few anomalous patterns inside the set of teaching patterns. These patterns would cause bad initialization results if they accidentally were selected as a center. By adding a small amount of noise, the negative effect caused by anomalous patterns can be lowered. However, if an exact interpolation is to be performed no modification of centers may be allowed.

The next initialization step is to set the free parameter p of the base function h, i.e. the bias of the hidden neurons. In order to do this, the initialization parameter *bias (p)* is directly copied into the bias of all hidden neurons. The setting of the bias is highly related to the base function h used and to the properties of the teaching patterns. When the Gaussian function is used, it is recommended to choose the value of the bias so that 5–10% of all hidden neurons are activated during propagation of every single teaching pattern. If the bias is chosen too small, almost all hidden neurons are uniformly activated during propagation. If the bias is chosen too large, only that hidden neuron is activated whose center vector corresponds to the currently applied teaching pattern.

Now the expensive initialization of the links between hidden and output layer is actually performed. In order to do this, the following formula which was already presented above is applied:

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

The initialization parameter 3 (*smoothness*) represents the value of λ in this formula. The matrices have been extended to allow an automatic computation of an additional constant value. If there is more than one neuron inside the output layer, the following set of functions results:

$$f_j(\vec{x}) = \sum_{i=1}^{K} c_{i,j} h_i(\vec{x}) + b_j$$

The bias of the output neuron(s) is directly set to the calculated value of b (b_j). Therefore, it is necessary to choose an activation function for the output neurons that uses the bias of the neurons. In the current version of SNNS, the functions `Act_Logistic` and `Act_IdentityPlusBias` implement this feature.

The activation functions of the output units lead to the remaining two initialization parameters. The initialization procedure assumes a linear activation of the output units. The link weights are calculated so that the weighted sum of the hidden neurons equals the teaching output. However, if a sigmoid activation function is used, which is recommended for pattern recognition tasks, the activation function has to be considered during initialization. Ideally, the supposed input for the activation function should be computed with the inverse activation function depending on the corresponding teaching output. This input value would be associated with the vector $\vec{y}$ during the calculation of weights. Unfortunately, the inverse activation function is unknown in the general case.

The first and second initialization parameters (*0_scale*) and (*1_scale*) are a remedy for this dilemma. They define the two control points of a piecewise linear function which approximates the activation function. *0_scale* and *1_ scale* give the net inputs of the output units which produce the teaching outputs 0 and 1. If, for example, the linear activation function `Act_IdentityPlusBias` is used, the values 0 and 1 have to be used. When using the logistic activation function `Act_Logistic`, the values -4 and 4 are recommended. If the bias is set to 0, these values lead to a final activation of 0.018 (resp. 0.982). These are comparatively good approximations of the desired teaching outputs 0 and 1. The implementation interpolates linearly between the set values of *0_scale* and *1_scale*. Thus, also teaching values which differ from 0 and 1 are mapped to corresponding input values.

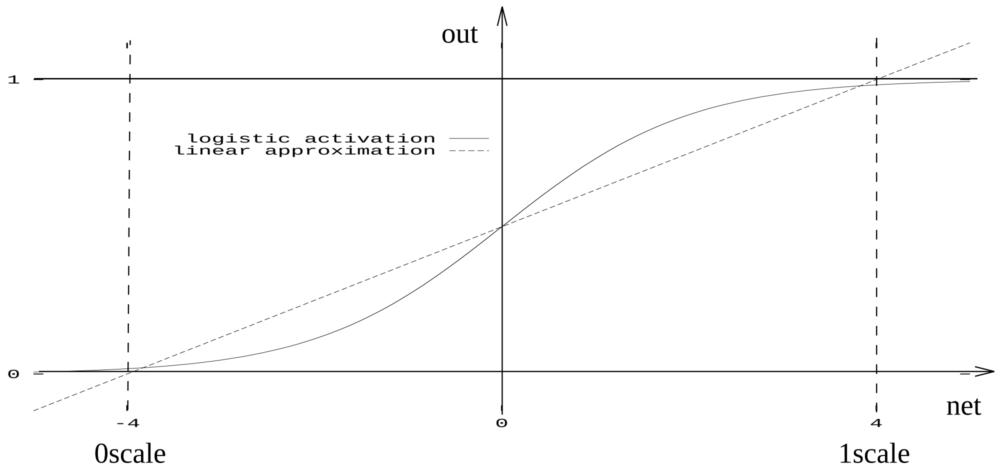

Figure 9.8: Relation between teaching output, input value and logistic activation

Figure 9.8 shows the activation of an output unit under use of the logistic activation function. The scale has been chosen in such a way, that the teaching outputs 0 and 1 are mapped to the input values -2 and 2.

The optimal values used for *0_scale* and *1_scale* can not be given in general. With the logistic activation function large scaling values lead to good initialization results, but interfere with the subsequent training, since the logistic function is used mainly in its very flat parts. On the other hand, small scaling values lead to bad initialization results, but produce good preconditions for additional training.

RBF_Weights_Kohonen One disadvantage of the above initialization procedure is the very simple selection of center vectors from the set of teaching patterns. It would be favorable if the center vectors would homogeneously cover the space of teaching patterns. `RBF_Weights_Kohonen` allows a self–organizing training of center vectors. Here, just as the name of the procedure already tells, the self–organizing maps of Kohonen are used (see [Was89]). The simplest version of Kohonen's maps has been implemented. It works as follows:

One precondition for the use of Kohonen maps is that the teaching patterns have to be normalized. This means, that they represent vectors with length 1. K patterns have to be selected from the set of n teaching patterns acting as starting values for the center vectors. Now the scalar product between one teaching pattern and each center vector is computed. If the vectors are normalized to length 1, the scalar product gives a measure for the distance between the two multiplied vectors. Now the center vector is determined whose distance to the current teaching pattern is minimal, i.e. whose scalar product is the largest one. This center vector is moved a little bit in the direction of the current teaching pattern:

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

This procedure is repeated for all teaching patterns several times. As a result, the center vectors adapt the statistical properties of the set of teaching patterns.

The resp. meanings of the three initialization parameters are:

1. *learn cycles*: determines the number of iterations of the Kohonen training for all teaching patterns. If 0 epochs are specified only the center vectors are set, but no training is performed.
2. *learning rate* α: It should be picked between 0 and 1. A learning rate of 0 leaves the center vectors unchanged. Using a learning rate of 1 replaces the selected center vector by the current teaching pattern.
3. *shuffle*: Determines the selection of initial center vectors at the beginning of the procedure. A value of 0 leads to the even selection already described for `RBF_Weights`. Any value other than 0 causes a random selection of center vectors from the set of teaching patterns.

Note, that the described initialization procedure initializes only the center vectors (i.e. the link weights between input and hidden layer). The bias values of the neurons have to be set manually using the graphical user interface. To perform the final initialization of missing link weights, another initialization procedure has been implemented.

RBF_Weights_Redo This initialization procedure influences only the link weights between hidden and output layer. It initializes the network as well as possible by taking the bias and the center vectors of the hidden neurons as a starting point. The center vectors can be set by the previously described initialization procedure. Another possibility is to create the center vectors by an external procedure, convert these center vectors into a SNNS pattern file and copy the patterns into the corresponding link weights by using the previously described initialization procedure. When doing this, Kohonen training must not be performed of course.

The effect of the procedure `RBF_Weights_Redo` differs from `RBF_Weights` only in the way that the center vectors and the bias remain unchanged. As expected, the last two initialization parameters are omitted. The meaning and effect of the remaining three parameters is identical with the ones described in `RBF_Weights`.

9.10.2.3 Learning Functions

Because of the special activation functions used for radial basis functions, a special learning function is needed. It is impossible to train networks which use the activation functions `Act_RBF_`... with backpropagation. The learning function for radial basis functions implemented here can only be applied if the neurons which use the special activation functions are forming the hidden layer of a three layer feedforward network. Also the neurons of the output layer have to pay attention to their bias for activation.

The name of the special learning function is `RadialBasisLearning`. The required parameters are:

1. η_1 *(centers)*: the learning rate used for the modification $\Delta\vec{t_j}$ of center vectors according to the formula $\Delta\vec{t_j} = -\eta_1 \frac{\partial E}{\partial \vec{t_j}}$.
2. η_2 *(bias p)*: learning rate used for the modification of the parameters p of the base function. p is stored as bias of the hidden units and is trained by the following formula $\Delta p_j = -\eta_2 \frac{\partial E}{\partial p_j}$.
3. η_3 *(weights)*: learning rate which influences the training of all link weights that are leading to the output layer as well as the bias of all output neurons.

$$\Delta c_{j,k} = -\eta_3 \frac{\partial E}{\partial c_{j,k}} \quad , \quad \Delta d_{i,k} = -\eta_3 \frac{\partial E}{\partial d_{i,k}} \quad , \quad \Delta b_k = -\eta_3 \frac{\partial E}{\partial b_k}$$

4. *delta max.*: To prevent an overtraining of the network the maximally tolerated error in an output unit can be defined. If the actual error is smaller than *delta max.* the corresponding weights are not changed. Common values range from 0 to 0.3.
5. *momentum*: momentum term during training, after the formula $\Delta g_{t+1} = -\eta \frac{\partial E}{\partial g} + \mu \Delta g_t$. The momentum–term is usually chosen between 0.8 and 0.9.

The learning rates η_1 to η_3 have to be selected very carefully. If the values are chosen too large (like the size of values for backpropagation) the modification of weights will be too extensive and the learning function will become unstable. Tests showed, that the learning procedure becomes more stable if only one of the three learning rates is set to a value bigger than 0. Most critical is the parameter *bias (p)*, because the base functions are fundamentally changed by this parameter.

Tests also showed that the learning function working in batch mode is much more stable than in online mode. Batch mode means that all changes become active not before all learning patterns have been presented once. This is also the training mode which is recommended in the literature about radial basis functions. The opposite of batch mode is known as online mode, where the weights are changed after the presentation of every single teaching pattern. Which mode is to be used can be defined during compilation of SNNS. The online mode is activated by defining the C macro `RBF_INCR_LEARNING` during compilation of the simulator kernel, while batch mode is the default.

9.10.3 Building a Radial Basis Function Application

As a first step, a three–layer feedforward network must be constructed with full connectivity between input and hidden layer and between hidden and output layer. Either the graphical editor or the tool `BIGNET` (both built into SNNS) can be used for this purpose.

The output function of all neurons is set to `Out_Identity`. The activation function of all hidden layer neurons is set to one of the three special activation functions `Act_RBF_`... (preferably to `Act_RBF_Gaussian`). For the activation of the output units, a function is needed which takes the bias into consideration. These functions are `Act_Logistic` and `Act_IdentityPlusBias`.

The next step consists of the creation of teaching patterns. They can be generated manually using the graphical editor, or automatically from external data sets by using an appropriate conversion program. If the initialization procedure `RBF_Weights_Kohonen` is going to be used, the center vectors should be normalized to length 1, or to equal length.

It is necessary to select an appropriate bias for the hidden units before the initialization is continued. Therefore, the link weights between input and hidden layer are set first, using the procedure `RBF_Weights_Kohonen` so that the center vectors which are represented by the link weights form a subset of the available teaching patterns. The necessary initialization parameters are: *learn cycles* = 0, *learning rate* = 0.0, *shuffle* = 0.0. Thereby teaching patterns are used as center vectors without modification.

To set the bias, the activation of the hidden units is checked for different teaching patterns by using the button TEST of the SNNS control panel. When doing this, the bias of the hidden neurons have to be adjusted so that the activations of the hidden units are as diverse as possible. Using the Gaussian function as base function, all hidden units are uniformly highly activated, if the bias is chosen too small (the case bias = 0 leads to an activation of 1 of all hidden neurons). If the bias is chosen too large, only the unit is activated whose link weights correspond to the current teaching pattern. A useful procedure to find the right bias is to first set the bias to 1, and then to change it uniformly depending on the behavior of the network. One must take care, however, that the bias does not become negative, since some implemented base functions require the bias to be positive. The optimal choice of the bias depends on the dimension of the input layer and the similarity among the teaching patterns.

After a suitable bias for the hidden units has been determined, the initialization procedure `RBF_Weights` can be started. Depending on the selected activation function for the output layer, the two *scale* parameters have to be set (see page 157). When `Act_IdentityPlus-Bias` is used, the two values 0 and 1 should be chosen. For the logistic activation function `Act_Logistic` the values -4 and 4 are recommended (also see figure 9.8). The parameters *smoothness* and *deviation* should be set to 0 first. The *bias* is set to the previously determined value. Depending on the number of teaching patterns and the number of hidden neurons, the initialization procedure may take rather long to execute. Therefore, some processing comments are printed on the terminal during initialization.

After the initialization has finished, the result may be checked by using the TEST button. However, the exact network error can only be determined by the teaching function. Therefore, the learning function `RadialBasisLearning` has to be selected first. All learning parameters are set to 0 and the number of learning cycles (CYCLES) is set to 1. After pressing the button ALL, the learning function is started. Since the learning parameters are set to 0, no changes inside the network will occur. After the presentation of all available teaching patterns, the actual error is printed to the terminal. As usual, the error is defined as the sum of squared errors of all output units (see formula 9.3). Under certain conditions it can be possible that the error becomes very large. This is mostly due to numerical problems. A poorly selected bias, for example, has shown to be a difficult starting point for the initialization. Also, if the number of teaching patterns is less than or equal to the number of hidden units a problem arises. In this case the number of unknown weights plus unknown bias values of output units exceeds the number of teaching patterns, i.e. there are more unknown parameters to be calculated than equations available. One or more neurons less inside the hidden layer then reduces the error considerably.

After the first initialization it is recommended to save the current network to test the possibilities of the learning function. It has turned out that the learning function becomes quickly unstable if too large learning rates are used. It is recommended to first set only one of the three learning rates (*centers, bias (p), weights*) to a value larger than 0 and to check the sensitivity of the learning function on this single learning rate. The use of the parameter *bias (p)* is exceptionally critical because it causes serious changes of the base function. If the bias of any hidden neuron is getting negative during learning, an appropriate message is printed to the terminal. In that case, a continuing meaningful training is impossible and the network should be reinitialized.

Immediately after initialization it is often useful to train only the link weights between hidden and output layer. Thereby the numerical inaccuracies which appeared during initialization are corrected. However, an optimized total result can only be achieved if also the center vectors are trained, since they might have been selected disadvantageously.

The initialization procedure used for direct link weight calculation is unable to calculate the weights between input and output layer. If such links are present, the following procedure is recommended: Even before setting the center vectors by using `RBF_Weights_Kohonen`, and before searching an appropriate bias, all weights should be set to random values between -0.1 and 0.1 by using the initialization procedure `Randomize_Weights`. Thereby, all links between input and output layer are preinitialized. Later on, after executing the procedure `RBF_Weights`, the error of the network will still be relatively large, because the above mentioned links have not been considered. Now it is easy to train these weights by only using the teaching parameter *weights* during learning.

9.11 Dynamic Decay Adjustment for RBFs (RBF–DDA)

9.11.1 The Dynamic Decay Adjustment Algorithm

The Dynamic Decay Adjustment (DDA–)Algorithm is an extension of the RCE-Algorithm (see [Hud92, RCE82]) and offers easy and constructive training for Radial Basis Function Networks. RBFs trained with the DDA-Algorithm often achieve classification accuracy comparable to Multi Layer Perceptrons (MLPs)[5] but training is significantly faster ([BD95]).

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

- The number of units in the **input layer** represents the dimensionality of the input space.
- The **hidden layer** contains the RBF units. Units are added in this layer during training. The input layer is fully connected to the hidden layer.
- Each unit in the **output layer** represents one possible class, resulting in an 1-of-n or binary coding. For classification a winner-takes-all approach is used, i.e. the output with the highest activation determines the class. Each hidden unit is connected to exactly one output unit.

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

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

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

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

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

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

An example of a full RBF-DDA is shown in figure 9.9. Note that there do not exist any shortcut connections between input and output units in an RBF-DDA.

In this illustration the weight vector that connects all input units to one hidden unit represents the centre of the Gaussian. The Euclidian distance of the input vector to this reference vector (or *prototype*) is used as an input to the Gaussian which leads to a local

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