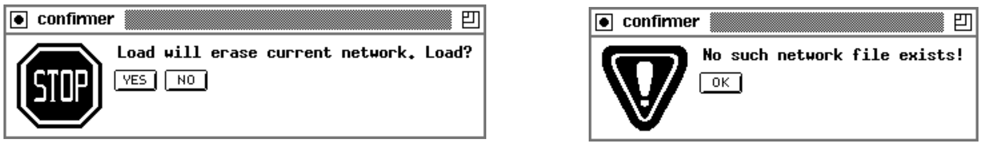

Figure 4.21: A normal confirmer and a message confirmer.

4.4 Parameters of the Learning Functions

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

- `Std_Backpropagation` ("Vanilla" Backpropagation),
  `BackpropBatch` and
  `TimeDelayBackprop`
  1. η: learning parameter, specifies the step width of the gradient descent.

     Typical values of η are $0.1 \ldots 1.0$. Some small examples actually train even faster with values above 1, like 2.0.
  2. d_{max}: the maximum difference $d_j = t_j - o_j$ between a teaching value t_j and an output o_j of an output unit which is tolerated, i.e. which is propagated back as $d_j = 0$. If values above 0.9 should be regarded as 1 and values below 0.1 as 0, then d_{max} should be set to 0.1. This prevents overtraining of the network.

     Typical values of d_{max} are 0, 0.1 or 0.2.
- `BackpropMomentum` (Backpropagation with momentum term and flat spot elimination):
  1. η: learning parameter, specifies the step width of the gradient descent.

     Typical values of η are $0.1 \ldots 1.0$. Some small examples actually train even faster with values above 1, like 2.0.
  2. μ: momentum term, specifies the amount of the old weight change (relative to 1) which is added to the current change.

     Typical values of μ are $0 \ldots 1.0$.
  3. c: flat spot elimination value, a constant value which is added to the derivative of the activation function to enable the network to pass flat spots of the error surface.

     Typical values of c are $0 \ldots 0.25$, most often 0.1 is used.
  4. d_{max}: the maximum difference $d_j = t_j - o_j$ between a teaching value t_j and an output o_j of an output unit which is tolerated, i.e. which is propagated back as $d_j = 0$. See above.

  The general formula for Backpropagation used here is

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

- `BackpropWeightDecay` (Backpropagation with Weight Decay)

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

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

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

  3. d_{min}: the minimum weight that is tolerated for a link. All links with a smaller weight will be pruned.

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

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

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

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

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

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

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

- `Quickprop`:

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

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

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

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

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

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

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

- `QuickpropThroughTime (QPTT)`:
  1. η: learning parameter, specifies the step width of the gradient descent.

     Typical values of η for QPTT are $0.005 \ldots 0.1$.
  2. μ: maximum growth parameter, specifies the maximum amount of weight change (relative to 1) which is added to the current change

     Typical values of μ are $1.2 \ldots 1.75$.
  3. ν: weight decay term to shrink the weights.

     Typical values of ν are $0.0005 \ldots 0.00005$.
  4. backstep: the number of quickprop steps back in time. QPTT stores a sequence of all unit activations while input patterns are applied. The activations are stored in a first-in-first-out queue for each unit.

     The largest backstep value supported is 10.
- `Counterpropagation`:
  1. α: learning parameter of the Kohonen layer.

     Typical values of α for Counterpropagation are $0.1 \ldots 0.7$.
  2. β: learning parameter of the Grossberg layer.

     Typical values of β are $0 \ldots 1.0$.
  3. θ: threshold of a unit.

     We often use a value θ of 0.
- `Backpercolation 1`:
  1. λ: global error magnification. This is the factor in the formula $\epsilon = \lambda(t - o)$, where ϵ is the internal activation error of a unit, t is the teaching input and o the output of a unit.

     Typical values of λ are 1. Bigger values (up to 10) may also be used here.
  2. θ: If the error value ξ drops below this threshold value, the adaption according to the Backpercolation algorithm begins. ξ is defined as:

     $$\xi = \frac{1}{pN} \sum^{p} \sum^{N} \mid o - \phi \mid$$
  3. d_{max}: the maximum difference $d_j = t_j - o_j$ between a teaching value t_j and an output o_j of an output unit which is tolerated, i.e. which is propagated back as $d_j = 0$. See above.

- `Dynamic Learning Vector Quantization (DLVQ)`:
  1. η^+: learning rate, specifies the step width of the mean vector $\vec{\mu_A}$, which is nearest to a pattern $\vec{x_A}$, towards this pattern. Remember that $\vec{\mu_A}$ is moved only, if $\vec{x_A}$ is not assigned to the correct class w_A. A typical value is 0.03.
  2. η^-: learning rate, specifies the step width of a mean vector $\vec{\mu_B}$, to which a pattern of class w_A is falsely assigned to, away from this pattern. A typical value is 0.03. Best results can be achieved, if the condition $\eta^+ = \eta^-$ is satisfied.
  3. Number of cycles you want to train the net before additive mean vectors are calculated.
- `RadialBasisLearning`:
  1. *centers*: determines the learning rate η_1 used for the modification of center vectors.
  2. *bias (p)*: determines the learning rate η_2, used for the modification of the parameters p of the base function. p is stored as bias of the hidden units.
  3. *weights*: influences the training of all link weights that are leading to the output layer as well as the training of the bias of all output neurons.
  4. *delta max.*: If the actual error is smaller than the maximum allowed error (*delta max.*) the corresponding weights are not changed.
  5. *momentum*:influences the amount of the momentum–term during training.
- `RadialBasisLearning with Dynamic Decay Adjustment`:
  1. θ^+: positive threshold. To commit a new prototype, none of the existing RBFs of the correct class may have an activation above θ^+
  2. θ^-:negative threshold. During shrinking no RBF unit of a conflicting class is allowed to have an activation above θ^-.
  3. n: the maximum number of RBF units to be diplayed in one row. This item allows the user to control the appearance of the network on the screen and has no influence on the performance.
- `ART1`
  1. ρ: vigilance parameter. If the quotient of active F_1 units divided by the number of active F_0 units is below ρ, an ART reset is performed.
- `ART2`
  1. ρ: vigilance parameter. Specifies the minimal length of the error vector $\mathbf{r}$ (units r_i).
  2. a: Strength of the influence of the lower level in F_1 by the middle level.
  3. b: Strength of the influence of the middle level in F_1 by the upper level.

4. c: Part of the length of vector $\mathbf{p}$ (units p_i) used to compute the error.
5. Θ: Threshold for output function f of units x_i and q_i.

- `ARTMAP`

1. $\overline{\rho^a}$: vigilance parameter for ART^a subnet. (quotient $\frac{|F_1^a|}{|F_0^a|}$)
2. ρ^b: vigilance parameter for ART^b subnet. (quotient $\frac{|F_1^b|}{|F_0^b|}$)
3. ρ: vigilance parameter for inter ART reset control. (quotient $\frac{|F^{ab}|}{|F_2^b|}$)

- `RPROP (resilient propagation)`

1. $delta_0$: starting values for all $\triangle_{ij}$. Default value is 0.1.
2. $delta_{max}$: the upper limit for the update values $\triangle_{ij}$.The default value of $\triangle_{max}$ is 50.0.
3. α: the weight-decay determines the relationship between the output error and to reduction in the size of the weights.*Important:* Please note that the weight decay parameter α denotes the exponent, to allow comfortable input of very small weight-decay. A choice of the third learning parameter $\alpha = 4$ corresponds to a ratio of weight decay term to output error of $1 : 10000 (1 : 10^4)$.

- `Cascade Correlation (CC)` and
  `Recurrent Cascade Correlation (RCC)`

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

  – Backpropagation (in CC or RCC):

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

  The general formula for this learning function is:

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

The slopes $\partial E/\partial w_{ij}$ and $-\partial C/\partial w_{ij}$ are abbreviated by S. This abbreviation is valid for all embedded functions. By changing the sign of the gradient value $\partial C/\partial w_{ij}$, the same learning function can be used to maximize the covariance and to minimize the error.

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

  The weight change is computed by:

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

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

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

  The formula used is:

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

- `Kohonen`

  1. *h(0):* Adaptation height. The initial adaptation height can vary between 0 and 1. It determines the overall adaptation strength.
  2. *r(0):* Adaptation radius. The initial adaptation radius $r(0)$ is the radius of the neighborhood of the winning unit. All units within this radius are adapted. Values should range between 1 and the size of the map.
  3. *mult_H:* Decrease factor. The adaptation height decreases monotonically after the presentation of every learning pattern. This decrease is controlled by the decrease factor mult_H: $h(t+1) := h(t) \cdot mult\_H$
  4. *mult_R:* Decrease factor. The adaptation radius also decreases monotonically after the presentation of every learning pattern. This second decrease is controlled by the decrease factor mult_R: $r(t+1) := r(t) * mult\_R$
  5. *h:* Horizontal size. Since the internal representation of a network doesn't allow to determine the 2-dimensional layout of the grid, the horizontal size in units must be provided for the learning function. It is the same value as used for the creation of the network.

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

  1. *n:* learning parameter, specifies the step width of the gradient descent. In [RM86] Rumelhart and McClelland use 0.01, although values less than 0.03 are generally acceptable.
  2. *Ncycles:* number of update cycles, specifies how many times a pattern is propagated through the network before the learning rule is applied. This parameter must be large enough so that the network is relatively stable after the set number of propagations. A value of 50 is recommended as a baseline. Increasing the value of this parameter increases the accuracy of the network but at a cost of processing time. Larger networks will probably require a higher setting of Ncycles.

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

- `Hebbian Learning`

  1. *n:* learning parameter, specifies the step width of the gradient descent. Values less than (1 / number of nodes) are recommended.
  2. *Wmax:* maximum weight strength, specifies the maximum absolute value of weight allowed in the network. A value of 1.0 is recommended, although this should be lowered if the network experiences explosive growth in the weights and activations. Larger networks will require lower values of Wmax.
  3. *count:* number of times the network is updated before calculating the error.

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

- `Monte-Carlo:`

  1. Min: lower limit of weights and biases. Typical values are $-10.0 \ldots -1.0$.
  2. Max: upper limit of weights and biases. Typical values are $1.0 \ldots 10.0$.

- `Simulated_Annealing_SS_error`,
  `Simulated_Annealing_WTA_error` and
  `Simulated_Annealing_WWTA_error:`

  1. Min: lower limit of weights and biases. Typical values are $-10.0 \ldots -1.0$.
  2. Max: upper limit of weights and biases. Typical values are $1.0 \ldots 10.0$.
  3. T_0: learning parameter, specifies the Simulated Annealing start temperature . Typical values of T_0 are $1.0 \ldots 10.0$.
  4. deg: degradation term of the temperature: $T_{new} = T_{old} \cdot deg$ Typical values of deg are $0.99 \ldots 0.99999$.

- `Scaled Conjugate Gradient (SCG)`

  All of the following parameters are non-critical, i.e. they influence only the speed of convergence, not whether there will be success or not.

  1. σ_1. Should satisfy $0 < \sigma_1 \leq 10^{-4}$. If 0, will be set to 10^{-4};
  2. λ_1. Should satisfy $0 < \lambda_1 \leq 10^{-6}$. If 0, will be set to 10^{-6};
  3. Δ_{max}. See standard backpropagation. Can be set to 0 if you don't know what to do with it;
  4. ϵ_1. Depends on the floating-point precision. Should be set to 10^{-8} (simple precision) or to 10^{-16} (double precision). If 0, will be set to 10^{-8}.

4.5 Update Functions

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

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

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

All these functions receive their input from the five update parameter fields in the control panel. See figure 4.11

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

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

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

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

ART1_Stable

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

ART1_Synchronous

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

ART2_Stable

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

ART2_Synchronous

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

ARTMAP_Stable

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

ARTMAP_Synchronous

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

Auto_Synchronous

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

BAM_Order

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

BBTT_Order

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

CC_Order

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

CounterPropagation

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

Dynamic_LVQ

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

Hopfield_Fixed_Act

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

Hopfield_Synchronous

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

JE_Order

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

JE_Special

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

Kohonen_Order

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

RCC_Order

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

Random_Order

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

Random_Permutation

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

Serial_Order

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

Synchronous_Order

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

TimeDelay_Order

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

Topological_Order

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

4.6 Initialization Functions

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

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

The following initialization functions are available:

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

All these functions receive their input from the five init parameter fields in the control panel. See figure 4.11

Here is a short description of the different initialization functions:

ART1_Weights

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

ART2_Weights

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

ARTMAP_Weights

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

CC_Weights

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

ClippHebb

The `ClippHebb` algorithm is almost the same as the `Hebb` algorithm, the only difference is that all weights can only be set to 1 and 0. After the activation for the neurons is calculated, all weights > 1 will be set to 1. As mentioned in 4.6 the `ClippHebb` algorithm is a learning algorithm.

CPN_Rand_Pat

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

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

CPN_Weights_v3.2

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

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

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

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

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

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

CPN_Weights_v3.3

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

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

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

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

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

DLVQ_Weights

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

Hebb

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

Hebb_FixAct

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

- The implemented net is an autoassociative net
  The neurons of an autoassociative net have to be input and output neurons at the same time. A Hopfield network would be an example for such a net.

- Fixed number of 1s
  The patterns which are to be saved have a fixed number of 1s.

The parameter h1 and h2 are required. Where h1 is the number of ones per pattern and h2 is the probable degree of distortion in percent. The parameters have to be inserted in field1 and field2. This initialization function should be used only in connection with the `Hopfield_Fixed_Act` update function. As mentioned in section 4.6 the `Hebb_FixAct` algorithm is a learning algorithm.

JE_Weights

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

- α, β : The weights of the forward connections are randomly chosen from the interval $[\alpha\,;\beta]$. α, β have to be provided in field1 and field2 of the init panel.

- λ : Weights of self recurrent links from context units to themselves. Simple Elman networks use $\lambda = 0$. λ has to be provided in field3 of the init panel.

- γ : Weights of other recurrent links to context units. This value is often set to 1.0. γ has to be provided in field4 of the init panel.

- ψ : Initial activation of all context units. ψ has to be provided in field5 of the init panel.

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

Kohonen_Weights_v3.2

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

Kohonen_Const

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

Kohonen_Rand_Pat

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

PseudoInv

The `PseudoInv` initialization function computes all weights with the help of the pseudo inverse weight matrix which is calculated with the algorithm of Greville. The formula for the weight calculation is: W = QS^+. Where S^+ is the 'Pseudoinverse' of the input vectors, Q are the output vectors and W are the desired weights of the net. The bias is not set and there are no parameters necessary. Please note that the calculated weights are usually odd. As mentioned in 4.6 the `PseudoInv` algorithm is a learning algorithm.

Randomize_Weights

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

Random_Weights_Perc

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

RBF_Weights

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

RBF_Weights_Kohonen

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

RBF_Weights_Redo

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

RCC_Weights

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

RM_Random_Weights

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

4.7 Creating and Editing Unit Prototypes and Sites

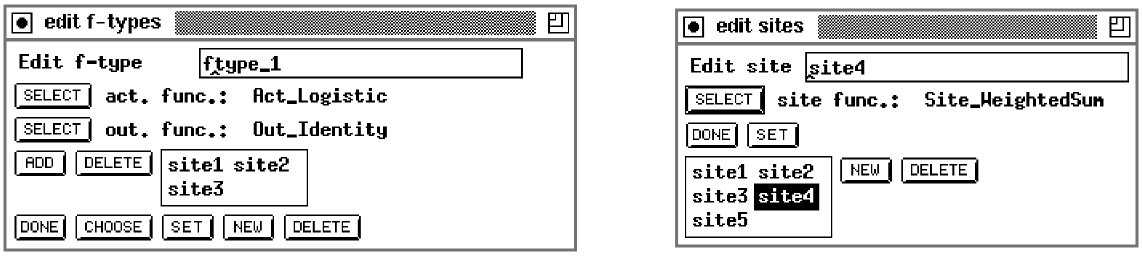

Figure 4.22: Edit panels for unit prototypes (f-types) and sites

Figure 4.22 shows the panels to edit unit prototypes (f-types) and sites. The change of the f-type is performed on all units of that type. Therefore, the functionality of all units assigned to an f-type can easily be changed. The elements in the panel have the following meaning:

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

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

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

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

Chapter 5

Handling Patterns with SNNS

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

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

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

5.1 Handling Pattern Sets

Although activations can be propagated through the network without patterns defined, learning can be performed only with patterns present. A set of patterns belonging to the same task is called a pattern set. Normally there are two dedicated pattern sets when dealing with a neural network. One for training the network (training pattern set), and one for testing purposes to see what the network has learned (test pattern set). In SNNS both of these (and more) can be kept in the simulator at the same time. They are loaded with the file browser (see chapter 4.3.2). The pattern set loaded last is made the current pattern set. All actions performed with the simulator refer only to, and affect only the current pattern set. To switch between pattern sets press the button $\boxed{\text{USE}}$ in the control panel (see figure 4.11 on page 39). It opens up a list of loaded pattern sets from which a new one can be selected. The name of the current pattern set is displayed to the right of the button. The name equals the name body of the loaded pattern file. If no pattern set is loaded, "Patternfile ?" is given as indication that no associated pattern file is defined.

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

5.2 Fixed Size Patterns

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

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

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

5.3 Variable Size Patterns

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

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

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

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

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

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

The meaning of the various lines is:

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

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

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

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

Line 0006 corresponds to line 0005 for the output pattern

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

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

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

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

Line 0010 corresponds to line 0008 for the output pattern.

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

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

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

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

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

Line 0215 corresponds to line 0012 for the output pattern.

Line 0216 corresponds to line 0013 for the output pattern.

Line 0217 corresponds to line 0014 for the output pattern.

Once the patterns are loaded into the simulator, their handling can be controlled by using the control panel. For the handling of variable size patterns an additional subpattern panel is provided. The handling of patterns is described in conjunction with the control panel description in section 4.3.3. All these explanations are intended for fixed sized patterns, but also hold true for variable size patterns so they are not repeated here.

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

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

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

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

[2] C is the value read from line 0005

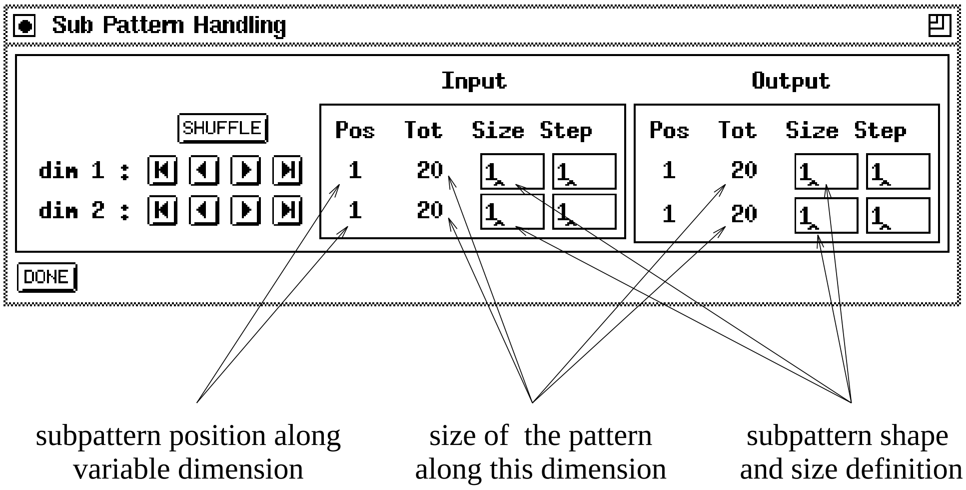

Figure 5.1: The handling panel for variable pattern sizes.

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

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

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

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

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

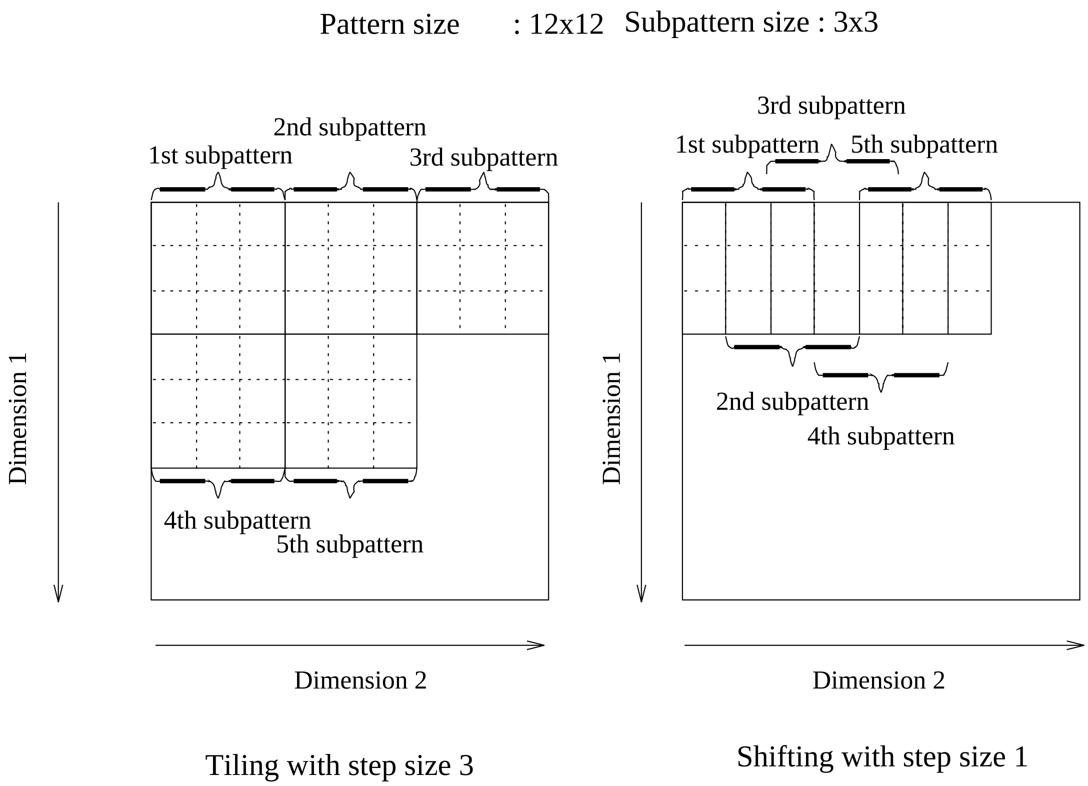

Figure 5.2: Tiling versus shifting of subpatterns.

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

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

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

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

Chapter 6

Graphical Network Editor

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

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

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

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

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

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

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

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

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

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

6.1 Editor Modes

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

Example (continued from above):

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

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

6.2 Selection

6.2.1 Selection of Units

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

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

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

Example (setting activations of a group of units):

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

6.2.2 Selection of Links

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

Example (deleting a group of links):

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

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

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

6.3 Use of the Mouse

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

- left mouse button:

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

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

- right mouse button:

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

- middle mouse button:

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

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

6.4 Short Command Reference

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

The following commands are possible within any command sequence

- `Quit`: quit a command
- `Return`: quit a command and return to normal mode (see chapter 6.1)
- `Help`: get help information. A help window pops up (see chapter 4.3.10)

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

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

1. Link Commands:

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

2. Site Commands:

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

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

3. Unit Commands:

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

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

4. Mode Commands:

- `Mode Units`: unit mode, shortens command sequence if one wants to work with unit commands only. All subsequences after the `Units` command are valid then
- `Mode Links`: analogous to `Mode Units`, but for link commands

5. Graphics Commands:

- `Graphics All`: redraws the local window
- `Graphics Complete`: redraws all windows
- `Graphics Direction`: draws all links from and to a unit with arrows in the local window
- `Graphics Links`: redraws all links in the local window
- `Graphics Move`: moves the origin of the local window such that the `Target` unit is displayed at the position of the mouse pointer
- `Graphics Origin`: moves the origin of the local window to the position indicated
- `Graphics Grid`: displays a graphic grid at the raster positions in the local window
- `Graphics Units`: redraws all units in the local window

6.5 Editor Commands

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

Links Make Clique (selection LINK : site-popup)

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

Units Set Activation (selection TARGET :)

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

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

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

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

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

The following keys are always possible within a command sequence:

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

A detailed description of the commands follows:

1. **Flags Safety** (:)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

   Links Copy Output (selection [unit] :)

   Links Copy All (selection [unit] :)

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

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

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

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

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

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

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

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

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

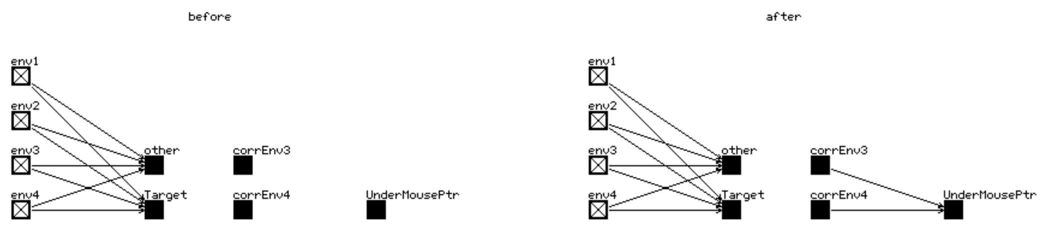

Figure 6.1: Example to `Links Copy Environment`

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

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

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

    Sites Copy with All links (selection SITE :)

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

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

    Units Unfreeze (selection :)

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

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

    Units Set Initial activation (selection TARGET :)

    Units Set Output (selection TARGET :)

    Units Set Bias (selection TARGET :)

    Units Set io-Type (selection : Popup)

    Units Set Function Activation (selection : Popup)

    Units Set Function Output (selection : Popup)

Units Set Function F-type (selection : Popup)

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

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

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

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

    Units Insert Target ([empty] TARGET :)

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

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

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

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

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

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

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

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

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

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

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

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

    Units Copy All

    Units Copy Input

    Units Copy Output

    Units Copy None

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

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

    Units Copy Structure All

    Units Copy Structure Input

    Units Copy Structure Output

    Units Copy Structure None

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

    Units Copy Structure Back binding

    Units Copy Structure Forward binding

    Units Copy Structure Double binding

    These commands are refinements of the general `Copy` command. Here, all links between the selected units are always copied as well. This means that the substructure is copied form the originals to the new units. On a copy without `Structure`