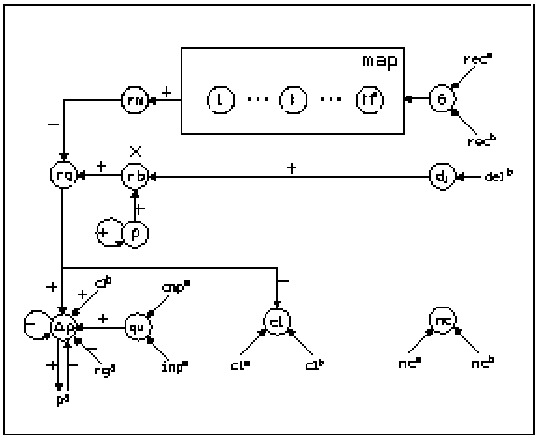

Figure 9.14: The MAP field with its control units.

The map field units represent the categories, onto which the ART^a classes are mapped[9]. The **G** unit is the MAP field gain unit. The units **rm** (reset map), **rb** (reset F_2^b), **rg** (reset general), ρ (vigilance) and $\mathbf{d}_1$ (delay 1) represent the inter-ART-reset control. $\Delta\rho$ and **qu** (quotient) have to realize the Match-Tracking-Mechanism and **cl** (classified) and **nc** (not classifiable) again show whether a pattern has been *classified* or was *not classifiable*.

9.12.3.2 Using ARTMAP Networks in SNNS

ARTMAP Initialization Function Since the trainable weights of an ARTMAP network are primarily the ones of the two ART1 networks ART^a and ART^b, it is easy to explain the ARTMAP initialization function `ARTMAP_Weights`. To use this function you have to select `ARTMAP_Weights` from the menu of the initialization functions. For `ARTMAP_Weights` you have to set the four parameters β^a, γ^a, β^b and γ^b. You can look up the meaning of each pair $\beta^?$, $\gamma^?$ in section 9.12.1.2, for the respective ART$^?$-part of the network.

ARTMAP Learning Function Select the ARTMAP learning function `ARTMAP` from the menu of the learning functions. Specify the three parameters $\overline{\rho^a}$, ρ^b and ρin the `LEARN` row of the control panel. Example values could be 0.7, 1.0, 1.0, and 0.0. $\overline{\rho^a}$ is the initial vigilance parameter for the ART^a-part of the net, which may be modified by the Match-Tracking-Mechanism. ρ^b is the vigilance parameter for the ART^b-part and ρ is the one for the Inter-ART-Reset control.

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

- `ARTMAP_Stable`

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

- ARTMAP_Synchronous.

ARTMAP_Stable is again used to propagate a pattern through the network until a stable state is reached, while ARTMAP_Synchronous does only perform one propagation step at a time. For both of the functions the parameters $\overline{\rho^a}$, ρ^b and ρ have to be specified in the line for update parameters of the control panel. The usage is the same as it is for ART1 and ART2 networks.

9.12.4 Topology of ART Networks in SNNS

The following tables are an exact description of the topology requirements for the ART models ART1, ART2 and ARTMAP. For ARTMAP the topologies of the two ART1-parts of the net are the same as the one shown in the ART1 table.

ART2

unit definition					connections	
unit name	top. type	activation function	output function	site names	target unit	target site
$\mathbf{inp}_i$	i	Act_Identity	Out_Identity		$\mathbf{w}_i$ $\mathbf{r}_i$	
$\mathbf{w}_i$	h	Act_ART2_Identity	Out_Identity		$\mathbf{x}_i$	
$\mathbf{x}_i$	h	Act_ART2_NormW	*signal function*[1]		$\mathbf{v}_i$	
$\mathbf{v}_i$	h	Act_ART2_Identity	Out_Identity		$\mathbf{u}_i$	
$\mathbf{u}_i$	h	Act_ART2_NormV	Out_Identity		$\mathbf{p}_i$ $\mathbf{w}_i$	
$\mathbf{p}_i$	h	Act_ART2_Identity	Out_Identity		$\mathbf{q}_i$ $\mathbf{r}_i$ $\mathbf{rec}_j\ \forall j$	
$\mathbf{q}_i$	h	Act_ART2_NormP	*signal function*[1]		$\mathbf{v}_i$	
$\mathbf{r}_i$	h	Act_ART2_NormIP	Out_Identity			
$\mathbf{rec}_j$	s	Act_ART2_Rec	Out_Identity		$\mathbf{p}_i\ \forall i$ $\mathbf{rst}_j$	
$\mathbf{rst}_j$	h	Act_ART2_Rst	Out_Identity		$\mathbf{rec}_j$	

[1] either Out_ART2_Noise_ContDiff or Out_ART2_Noise_PLin.

ARTMAP

site definition		
site name	site function	
ARTa_G	Site_at_least_1	
ARTb_G	Site_at_least_1	
ARTb_rb	Site_WeightedSum	
rho_rb	Site_WeightedSum	
npa_qu	Site_Reciprocal	
cmpa_qu	Site_WeightedSum	

unit definition					connections	
unit name	top. type	activation function	output function	site names	target unit	target site
map$_j$	h	Act_at_least_2	Out_Identity		**rm**	
cl	h	Act_at_least_2	Out_Identity			
nc	h	Act_at_least_1	Out_Identity			
G	h	Act_exactly_1	Out_Identity	ARTa_G ARTb_G	**map**$_j$ $\forall j$	
d$_1$	h	Act_Identity	Out_Identity		**rb**	ARTb_rb
rb	h	Act_Product	Out_Identity	ARTb_rb rho_rb	**rg**	
rm	h	Act_Identity	Out_Identity		**rg**	
rg	h	Act_less_than_0	Out_Identity		**drho** **cl**	
rho	h	Act_Identity	Out_Identity		**rho** **rb**	 rho_rb
qu	h	Act_Product	Out_Identity	inpa_qu cmpa_qu	**drho**	
drho	h	Act_ARTMAP_DRho	Out_Identity		**drho** **rho**^{a}	
inp$_i^a$	see ART1 table				**qu**	inpa_qu
cmp$_i^a$	see ART1 table				**qu**	cmpa_qu
rec$_i^a$	see ART1 table				**G**	ARTa_G
rec$_i^b$	see ART1 table				**G**	ARTb_G
del$_i^b$	see ART1 table				**d**$_1$	
cl^a	see ART1 table				**cl**	
cl^b	see ART1 table				**cl** **drho**	
nc^a	see ART1 table				**nc**	
nc^b	see ART1 table				**nc**	
rg^a	see ART1 table				**drho**	
rho^a	see ART1 table				**drho**	

ART1 and ART1-parts of ARTMAP (ART^a, ART^b)

site definition	
site name	site function
`rst_self`	`Site_WeightedSum`
`rst_signal`	`Site_at_least_2`
`inp_g1`	`Site_at_least_1`
`rec_g1`	`Site_at_most_0`
`inp_ri`	`Site_WeightedSum`
`rho_ri`	`Site_WeightedSum`

unit definition					**connections**	
unit name	top. type	activation function	output function	site-names	target unit	target site

inp$_i$	i	`Act_Identity`	`Out_Identity`		**cmp**$_i$
					g$_1$
					ri
					g$_2$
						
						`inp_g1`
						`inp_ri`

cmp$_i$	h	`Act_at_least_2`	`Out_Identity`		**rec**$_j$ $\forall j$ **rc**	
rec$_j$	s	`Act_Identity`	`Out_Identity`		**del**$_j$ **g**$_1$	 `rec_g1`
del$_j$	h	`Act_at_least_2`	`Out_Identity`		**cmp**$_i$ $\forall i$ **d**$_1$ **rst**$_j$	 `rst_signal`
d$_1$	h	`Act_at_least_1`	`Out_Identity`		**d**$_2$	
d$_2$	h	`Act_at_least_1`	`Out_Identity`		**d**$_3$	
d$_3$	h	`Act_at_least_1`	`Out_Identity`		**cl**	
rst$_j$	h	`Act_at_least_1`	`Out_Identity`	`rst_self` `rst_signal`	**rst**$_j$ **rec**$_j$	`rst_self`
cl	h	`Act_at_least_1`	`Out_Identity`			

nc	h	`Act_ART1_NC`
		for ARTMAP:
		`Act_ARTMAP_NCa`,
		`Act_ARTMAP_NCb`
			`Out_Identity`			

g$_1$	h	`Act_at_least_2`	`Out_Identity`	`inp_g1` `rec_g1`	**cmp**$_i$ $\forall i$	
g$_2$	h	`Act_at_most_0`	`Out_Identity`		**rec**$_j$ $\forall j$ **cl**	
ri	h	`Act_Product`	`Out_Identity`	`inp_ri` `rho_ri`	**rg**	
rc	h	`Act_Identity`	`Out_Identity`		**rg**	
rg	h	`Act_less_than_0`	`Out_Identity`		**rec**$_j$ $\forall j$ **rst**$_j$ $\forall j$ **cl**	 `rst_signal`
rho	h	`Act_Identity`	`Out_Identity`		**rho** **ri**	 `rho_ri`

9.13 Self-Organizing Maps (SOMs)

9.13.1 SOM Fundamentals

The Self-Organizing Map (SOM) algorithm of Kohonen, also called Kohonen feature map, is one of the best known artificial neural network algorithms. In contrast to most other algorithms in SNNS, it is based on unsupervised learning. SOMs are a unique class of neural networks, since they construct topology-preserving mappings of the training data where the location of a unit carries semantic information. Therefore, the main application of this algorithm is clustering of data, obtaining a two-dimensional display of the input space that is easy to visualize.

Self-Organizing Maps consist of two layers of units: A one dimensional input layer and a two dimensional competitive layer, organized as a 2D grid of units. This layer can neither be called hidden nor output layer, although the units in this layer are listed as hidden units within SNNS. Each unit in the competitive layer holds a weight (reference) vector, W_i, that, after training, resembles a different input pattern. The learning algorithm for the SOM accomplishes two important things:

1. clustering the input data
2. spatial ordering of the map so that similar input patterns tend to produce a response in units that are close to each other in the grid.

Before starting the learning process, it is important to initialize the competitive layer with normalized vectors. The input pattern vectors are presented to all competitive units in parallel and the best matching (nearest) unit is chosen as the winner. Since the vectors are normalized, the similarity between the normalized input vector X $= (x_i)$ and the reference units $W_j = (w_{ij})$ can be calculated using the dot product:

$$Net_j(t) = X \cdot W_j = \sum_{i=1}^{n} x_i(t) w_{ij}(t)$$

The vector[10] W_c most similar to X is the one with the largest dot product with X:

$$Net_c(t) = \max_j \{Net_j(t)\} = X \cdot W_c$$

The topological ordering is achieved by using a spatial neighborhood relation between the competitive units during learning. I.e. not only the best-matching vector, with weight W_c, but also its neighborhood [11] N_c, is adapted, in contrast to a basic competitive learning algorithm like LVQ:

[10] c will be used as index for the winning unit in the competitive layer throughout this text

[11] Neighborhood is defined as the set of units within a certain radius of the winner. So $N(1)$ would be the the eight direct neighbors in the 2D grid; $N(2)$ would be $N(1)$ plus the 16 next closest; etc.

$$\begin{aligned} \Delta w_{ij}(t) &= e_j(t) \cdot (x_i(t) - w_{ij}(t)) && \text{for } j \in N_c \\ \Delta w_{ij}(t) &= 0 && \text{for } j \notin N_c \end{aligned}$$

where

$e_j(t) = h(t) \cdot e^{-(d_j/r(t))^2}$ (Gaussian Function)
d_j: distance between W_j and winner W_c
$h(t)$: adaptation height at time t with $0 \le h(t) \le 1$
$r(t)$: radius of the spatial neighborhood N_c at time t

The adaption height and radius are usually decreased over time to enforce the clustering process.

See [Koh88] for a more detailed description of SOMs and their theory.

9.13.2 SOM Implementation in SNNS

SNNS originally was not designed to handle units whose location carry semantic information. Therefore some points have to be taken care of when dealing with SOMs. For learning it is necessary to pass the horizontal size of the competitive layer to the learning function, since the internal representation of a map is different from its appearance in the display. Furthermore it is not recommended to use the graphical network editor to create or modify any feature maps. The creation of new feature maps should be carried out with the **BIGNET (Kohonen)** creation tool (see chapter 7).

9.13.2.1 The KOHONEN Learning Function

SOM training in SNNS can be performed with the learning function `Kohonen`. It can be selected from the list of learning functions in the control panel. Five parameters have to be passed to this learning function:

- **Adaptation Height (Learning Height)** The initial adaptation height $h(0)$ can vary between 0 and 1. It determines the overall adaptation strength.
- **Adaptation Radius (Learning 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.
- **Decrease Factor mult_H** The adaptation height decreases monotonically after the presentation of every learning pattern. This decrease is controlled by the decrease factor mult_H: $h(t+1) := h(t) \cdot mult\_H$
- **Decrease Factor mult_R** The adaptation radius also decreases monotonically after the presentation of every learning pattern. This second decrease is controlled by the decrease factor mult_R: $r(t+1) := r(t) * mult\_R$

- **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.

Note: After each completed training the parameters adaption height and adaption radius are updated in the control panel to reflect their new values. So when training is started anew, it resumes at the point where it was stopped last. Both mult_H and mult_R should be in the range $(0, 1]$. A value of 1 consequently keeps the adaption values at a constant level.

9.13.2.2 The Kohonen Update Function

A special update function (`Kohonen_Order`) is also provided in SNNS. This function has to be used since it is the only one that takes care of the special ordering of units in the competitive layer. If another update is selected an "Error: Dead units in the network" may occur when propagating patterns.

9.13.2.3 The Kohonen Init Function

Before a SOM can be trained, its weights have to be initialized using the init function `Kohonen_Weights`. This init function first initializes all weights with random values between the specified borders. Then it normalizes all links on a per-unit basis. Some of the internal values needed for later training are also set here. It uses the same parameters as `CPN_Weights` (see section 9.5.2).

9.13.2.4 The Kohonen Activation Functions

The Kohonen learning algorithm does not use the activation functions of the units. Therefore it is basically unimportant which activation function is used in the SOM. For display and evaluation reasons, however, the two activation functions `Act_Component` and `Act_Euclid` have been implemented in SNNS. `Act_Euclid` copies the Euclidian distance of the unit from the training pattern to the unit activation, while `Act_Component` writes the weight to one specific component unit into the activation of the unit.

9.13.2.5 Building and Training Self-Organizing Maps

Since any modification of a Self-Organizing Map in the 2D display like the creation, deletion or movement of units or weights may destroy the relative position of the units in the map we strongly recommend to generate these networks only with the available **BIGNET (Kohonen)** tool. See also chapter 7 for detailed information on how to create networks. Outside xgui you can also use the tool **convert2snns**. Information on this program can be found in the respective README file in the directory SNNSv3.3/tools/doc. Note:

Any modification of the units after the creation of the network may result in undesired behavior!

To train a new feature map with SNNS, set the appropriate standard functions: select init function **KOHONEN_Weights**, update function **Kohonen_Order** and learning function **Kohonen**. Remember: There is no special activation function for Kohonen learning, since setting an activation function for the units doesn't affect the learning procedure. To visualize the results of the training, however, one of the two activation functions `Act_Euclid` and `Act_Componnent` has to be selected. For their semantics see section 9.13.2.6.

After providing patterns (ideally normalized) and assigning reasonable values to the learning function, the learning process can be started. To get a proper appearance of SOMs in the 2D-display set the **grid width** to 16 and turn off the unit labeling and link display in the display panel.

When a learning run is completed the adaption height and adaption radius parameters are automatically updated in the control panel to reflect the actual values in the kernel.

9.13.2.6 Evaluation Tools for SOMs

When the results of the learning process are to be analyzed, the tools described here can be used to evaluate the qualitative properties of the SOM. In order to provide this functionality, a special panel was added. It can be called from the manager panel by clicking the `KOHONEN` button and is displayed in figure 9.15. Yet, the panel can only be used in combination with the control panel.

Figure 9.15: The additional KOHONEN (control) panel

1. **Euclidian distance**
The distance between an input vector and the weight vectors can be visualized using a distance map. This function allows using the SOM as a classifier for arbitrary input patterns: Choose **Act_Euclid** as activation function for the hidden units, then use the TEST button in the control panel to see the distance maps of consecutive patterns. As green squares (big filled squares on B/W terminals) indicate high activations, green squares here mean big distances, while blue squares represent small distances. Note: The input vector is not normalized before calculating the distance to the competitive units. This doesn't affect the qualitative appearance of the distance maps, but offers the advantage of evaluating SOMs that were generated by different SOM-algorithms (learning without normalization). If the dot product as similarity measure is to be used select **Act_Identity** as activation function for the hidden units.

2. **Component maps**
   To determine the quality of the clustering for each component of the input vector use this function of the SOM analyzing tool. Due to the topology-preserving nature of the SOM algorithm, the component maps can be compared after printing, thereby detecting correlations between some components: Choose the activation function **Act_Component** for the hidden units. Just like displaying a pattern, component maps can be displayed using the **LAYER** buttons in the **KOHONEN** panel. Again, green squares represent large, positive weights.

3. **Winning Units**
   The set of units that came out as winners in the learning process can also be displayed in SNNS. This shows the distribution of patterns on the SOM. To proceed, turn on **units top** in the setup window of the display and select the **winner** item to be shown. New winning units will be displayed without deleting the existing, which enables tracing the temporal development of clusters while learning is in progress. The display of the winning units is refreshed by pressing the WINNER button again.

   Note: Since the winner algorithm is part of the KOHONEN learning function, the learning parameters must be set as if learning is to be performed.

9.14 Autoassociative Networks

9.14.1 General Characteristics

Autoassociative networks store single instances of items, and can be thought of in a way similar to human memory. In an autoassociative network each pattern presented to the network serves as both the input and the output pattern. Autoassociative networks are typically used for tasks involving pattern completion. During retrieval, a probe pattern is presented to the network causing the network to display a composite of its learned patterns most consistent with the new information.

Autoassociative networks typically consist of a single layer of nodes with each node representing some feature of the environment. However in SNNS they are represented by two layers to make it easier to compare the input to the output. The following section explains the layout in more detail[12].

Autoassociative networks must use the update function `RM_Synchronous` and the initialization function `RM_Random_Weights`. The use of others may destroy essential characteristics of the autoassociative network. Please note, that the update function `RM_Synchronous` needs as a parameter the number of iterations performed before the network output is computed. 50 has shown to be very suitable here.

All the implementations of autoassociative networks in SNNS report error as the sum of squared error between the input pattern on the world layer and the resultant pattern on

[12] For any comments or questions concerning the implementation of an autoassociative memory please refer to Jamie DeCoster at jamie@psych.purdue.edu

the learning layer after the pattern has been propagated a user defined number of times.

9.14.2 Layout of Autoassociative Networks

An autoassociative network in SNNS consists of two layers: A layer of **world** units and a layer of **learning** units. The representation on the world units indicates the information coming into the network from the outside world. The representation on the learning units represents the network's current interpretation of the incoming information. This interpretation is determined partially by the input and partially by the network's prior learning.

Figure 9.16 shows a simple example network. Each unit in the world layer sends input to exactly one unit in the learning layer. The connected pair of units corresponds to a single node in the typical representation of autoassociative networks. The link from the world unit always has a weight of 1.0, and is unidirectional from the world unit to the learning unit. The learning units are fully interconnected with each other.

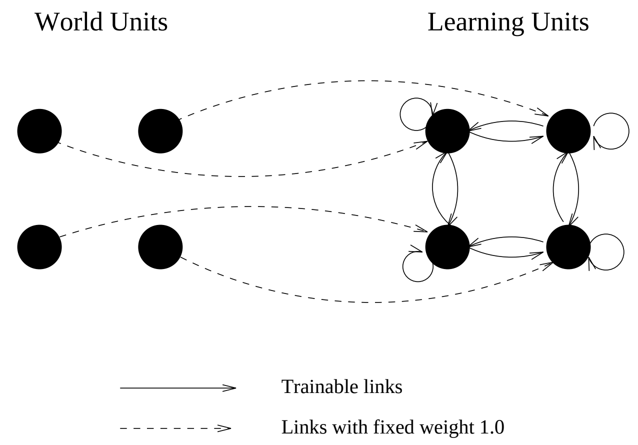

Figure 9.16: A simple Autoassociative Memory Network

The links between the learning units change according to the selected learning rule to fit the representation on the world units. The links between the world units and their corresponding learning units are not affected by learning.

9.14.3 Hebbian Learning

In Hebbian learning weights between learning nodes are adjusted so that each weight better represents the relationship between the nodes. Nodes which tend to be positive or negative at the same time will have strong positive weights while those which tend to be opposite will have strong negative weights. Nodes which are uncorrelated will have weights near zero.

The general formula for Hebbian learning is

$$\Delta w_{ij} = n * input_i * input_j$$

where:

n	is the learning rate
$input_i$	is the external input to node i
$input_j$	is the external input to node j

9.14.4 McClelland & Rumelhart's Delta Rule

This rule is presented in detail in chapter 17 of [RM86]. In general the delta rule outperforms the Hebbian learning rule. The delta rule is also less likely so produce explosive growth in the network. For each learning cycle the pattern is propagated through the network ncycles (a learning parameter) times after which learning occurs. Weights are updated according to the following rule:

$$\Delta w_{ij} = n * d_i * a_j$$

where:

n	is the learning rate
a_j	is the activation of the source node
d_i	is the error in the destination node. This error is defined as the external input - the internal input.

In their original work McClelland and Rumelhart used an unusual activation function:

```
for unit i,
    if neti > 0
       delta ai = E * neti * (1 - ai) - D * ai
    else
       delta ai = E * neti * (ai + 1) - D * ai
```

where:

$neti$	is the net input to i (external + internal)
E	is the excitation parameter (here set to 0.15)
D	is the decay parameter (here set to 0.15)

This function is included in SNNS as `ACT_RM`. Other activation functions may be used in its place.

9.15 Partial Recurrent Networks

9.15.1 Models of Partial Recurrent Networks

9.15.1.1 Jordan Networks

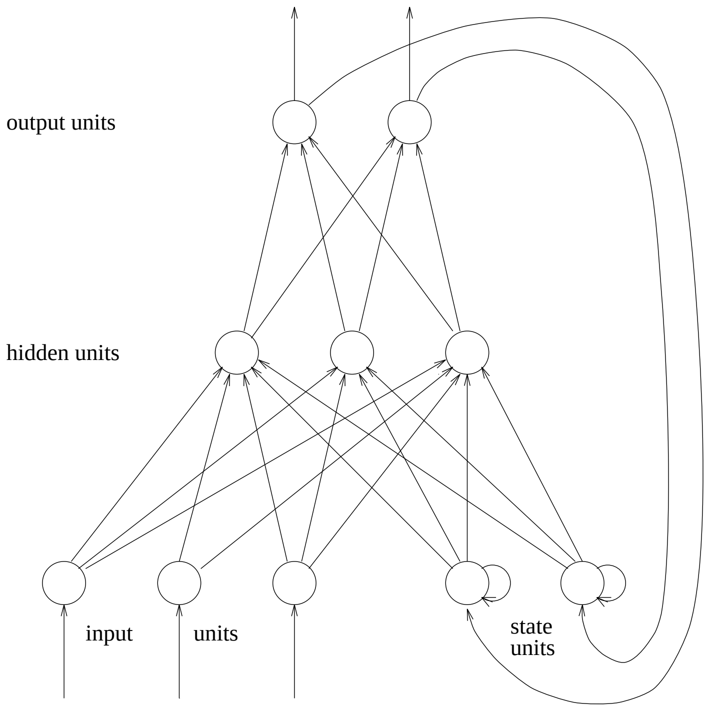

Figure 9.17: Jordan network

Literature: [Jor86b], [Jor86a]

9.15.1.2 Elman Networks

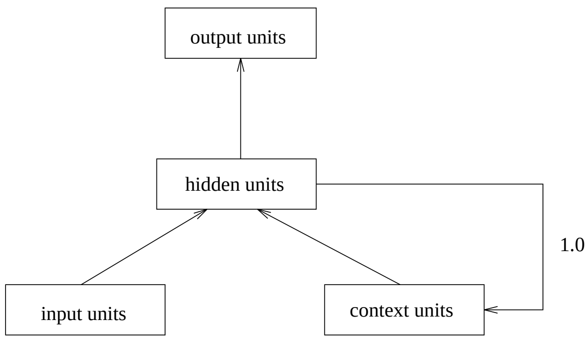

Figure 9.18: Elman network

Literature: [Elm90]

9.15.1.3 Extended Hierarchical Elman Networks

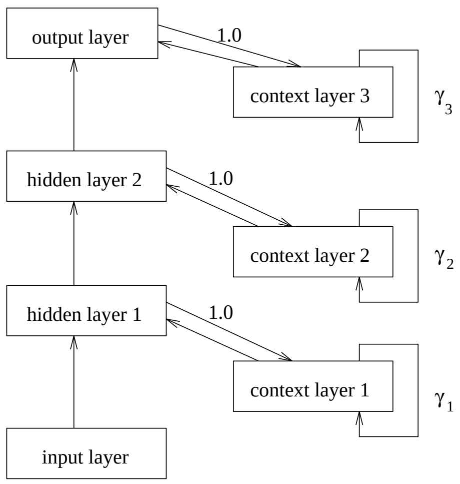

Figure 9.19: Extended Elman architecture

9.15.2 Working with Partial Recurrent Networks

In this subsection, the initialization, learning, and update functions for partial recurrent networks are described. These functions can not be applied to only the three network models described in the previous subsection. They can be applied to a broader class of partial recurrent networks. Every partial recurrent network, that has the following restrictions, can be used:

- If after the deletion of all context units and the links to and from them, the remaining network is a simple feedforward architecture with no cycles.
- Input units must not get input from other units.
- Output units may only have outgoing connections to context units, but not to other units.
- Every unit, except the input units, has to have at least one incoming link. For a context unit this restriction is already fulfilled when there exists only a self recurrent link. In this case the context unit receives its input only from itself.

In such networks all links leading to context units are considered as recurrent links.

Thereby the user has a lot of possibilities to experiment with a great variety of partial recurrent networks. E.g. it is allowed to connect context units with other context units.

Note: context units are realized as special hidden units. All units of type special hidden are assumed to be context units and are treated like this.

9.15.2.1 The Initialization Function JE_Weights

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 $[\alpha\,;\beta]$.
- λ : Weights of self recurrent links from context units to themselves. Simple Elman networks use $\lambda = 0$.
- γ : Weights of other recurrent links to context units. This value is often set to 1.0.
- ψ : Initial activation of all context units.

These values are to be set in the `INIT` line of the control panel in the order given above.

9.15.2.2 Learning Functions

By deleting all recurrent links in a partial recurrent network, a simple feedforward network remains. The context units have now the function of input units, i.e. the total network input consists of two components. The first component is the pattern vector, which was the only input to the partial recurrent network. The second component is a state vector. This state vector is given through the next–state function in every step. By this way the behavior of a partial recurrent network can be simulated with a simple feedforward network, that receives the state not implicitly through recurrent links, but as an explicit part of the input vector. In this sense, backpropagation algorithms can easily be modified for the training of partial recurrent networks in the following way:

1. Initialization of the context units. In the following steps, all recurrent links are assumed to be not existent, except in step 2(f).
2. Execute for each pattern of the training sequence the following steps:
   - input of the pattern and forward propagation through the network
   - calculation of the error signals of output units by comparing the computed output and the teaching output
   - back propagation of the error signals
   - calculation of the weight changes
   - only on–line training: weight adaption
   - calculation of the new state of the context units according to the incoming links
3. Only off–line training: weight adaption

In this manner, the following learning functions have been adapted for the training of partial recurrent networks like Jordan and Elman networks:

- `JE_BP`: Standard Backpropagation for partial recurrent networks

- `JE_BPMomentum`: Standard Backpropagation with Momentum–Term for partial recurrent networks
- `JE_Quickprop`: Quickprop for partial recurrent networks
- `JE_Rprop`: Rprop for partial recurrent networks

The parameters for these learning functions are the same as for the regular feedforward versions of these algorithms (see section 4.4) plus one special parameter.

For training a network with one of these functions a method called teacher forcing can be used. Teacher forcing means that during the training phase the output units propagate the teaching output instead of their produced output to successor units (if there are any). The new parameter is used to enable or disable teacher forcing. If the value is less or equal 0.0 only the teaching output is used, if it is greater or equal 1.0 the real output is propagated. Values between 0.0 and 1.0 yield a weighted sum of the teaching output and the real output.

9.15.2.3 Update Functions

Two new update functions have been implemented for partial recurrent networks:

- `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.
- `JE_Special`:
  This update function can be used for iterated long-term predictions. If the actual prediction value p_i is part of the next input pattern inp_{i+1} to predict the next value p_{i+1}, the input pattern inp_{i+1} can not be generated before the needed prediction value p_i is available. For long-term predictions many input patterns have to be generated in this manner. To generate these patterns manually means a lot of effort.

  Using the update function `JE_Special` these 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 new 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`.

9.16 Stochastic Learning Functions

The monte carlo method and simulated annealing are widely used algorithms for solving any kind of optimization problems. These stochastic functions have some advantages over other learning functions. They allow any net structure, any type of neurons and any type of error function. Even every neuron of a net could have another learning function and any number of links.

9.16.1 Monte-Carlo

Monte-Carlo learning is an easy way to determine weights and biases of a net. At every learning cycle all weights and biases are chosen by random in the Range $[Min, Max]$. Then the error is calculated as summed squared error of all patterns. If the error is lower than the previous best error, the weights and biases are stored. This method is not very efficient but useful for finding a good start point for another learning algorithm.

9.16.2 Simulated Annealing

Simulated annealing is a more sophisticated method for finding the global minima of a error surface. In contrast to monte carlo learning only one weight or bias is changed at a learning cycle. Dependant on the error development and a system temperature this change is accepted or rejected. One of the advantages of simulated annealing is that learning does not get stuck in local minima.

At the beginning of learning the temperature T is set to T_0. Each training cycle consists of the following four steps.

1. Change one weight or bias by random in the range $[Min, Max]$.
2. Calculate the net error as sum of the given error function for all patterns.
3. Accept change if the error decreased or if the error increased by ΔE with the probability p given by: $p = \exp\left(\frac{-\Delta E}{T}\right)$
4. Decrease the temperature: $T = T \cdot deg$

The three implemented simulated annealing functions only differ in the way the net error is calculated. Sim_Ann_SS calculates a summed squared error like the backpropagation learning functions; Sim_Ann_WTA calculates a winner takes all error; and Sim_Ann_WWTA calculates a winner takes all error and adds a term corresponding to the security of the winner takes all decision.

9.17 Scaled Conjugate Gradient (SCG)

SCG [Mol93] is a supervised learning algorithm for feedforward neural networks, and is a member of the class of conjugate gradient methods. Before describing SCG, we recall some key points concerning these methods. Eventually we will discuss the parameters (virtually none) and the complexity of SCG.

9.17.1 Conjugate Gradient Methods (CGMs)

They are general purpose second order techniques that help minimize goal functions of several variables, with sound theoretical foundations [P+88, Was95]. Second order means that these methods make use of the second derivatives of the goal function, while first-order techniques like standard backpropagation only use the first derivatives. A second order technique generally finds a better way to a (local) minimum than a first order technique, but **at a higher computational cost**.

Like standard backpropagation, CGMs iteratively try to get closer to the minimum. But while standard backpropagation always proceeds down the gradient of the error function, a conjugate gradient method will proceed in a direction which is **conjugate** to the directions of the previous steps. Thus the minimization performed in one step is not partially undone by the next, as it is the case with standard backpropagation and other gradient descent methods.

9.17.2 Main features of SCG

Let w_i be a vector from the space R^N, where N is the sum of the number of weights and of the number of biases of the network. Let E be the error function we want to minimize.

SCG differs from other CGMs in two points:

- each iteration k of a CGM computes $w_{k+1} = w_k + \alpha_k \cdot p_k$, where p_k is a new conjugate direction, and α_k is the size of the step in this direction. Actually α_k is a function of $E''(w_k)$, the Hessian matrix of the error function, namely the matrix of the second derivatives. In contrast to other CGMs which avoid the complex computation of the Hessian and approximate α_k with a **time-consuming** line search procedure, SCG makes the following simple approximation of the term s_k, a key component of the computation of α_k:

$$s_k = E''(w_k) \cdot p_k \approx \frac{E'(w_k + \sigma_k \cdot p_k) - E'(w_k)}{\sigma_k}, \ 0 < \sigma_k \ll 1$$

- as the Hessian is not always positive definite, which prevents the algorithm from achieving good performance, SCG uses a scalar λ_k which is supposed to regulate the indefiniteness of the Hessian. This is a kind of Levenberg-Marquardt method [P+88], and is done by setting:

$$s_k = \frac{E'(w_k + \sigma_k \cdot p_k) - E'(w_k)}{\sigma_k} + \lambda_k \cdot p_k$$

  and adjusting λ_k at each iteration. This is the main contribution of SCG to both fields of neural learning and optimization theory.

SCG has been shown to be considerably faster than standard backpropagation and than other CGMs [Mol93].

9.17.3 Parameters of SCG

As σ_k and λ_k are computed from their respective values at step $k-1$, SCG has two parameters, namely the initial values σ_1 and λ_1. Their values are not critical but should respect the conditions $0 < \sigma_1 \leq 10^{-4}$ and $0 < \lambda_1 \leq 10^{-6}$. Empirically Møller has shown that bigger values of σ_1 can lead to a slower convergence.

The third parameter is the usual quantity Δ_{max} (cf. standard backpropagation).

In SNNS, it is usually the responsibility of the user to determine when the learning process should stop. Unfortunately, the λ_k adaptation mechanism sometimes assigns too large values to λ_k when no more progress is possible. In order to avoid floating-point exceptions, we have added a termination criterion[13] to SCG. The criterion is taken from the CGMs presented in [P+88]: stop when

$$2 \cdot (|E(w_{k+1}) - E(w_k)|) \leq \epsilon_1 \cdot (|E(w_{k+1})| + |E(w_k)| + \epsilon_2)$$

ϵ_2 is a small number used to rectify the special case of converging to a function value of exactly zero. It is set to 10^{-10}. ϵ_1 is a tolerance depending of the floating-point precision Φ of your machine, and it should be set to Φ, which is usually equal to 10^{-8} (simple precision) or to 10^{-16} (double precision).

To summarize, there are four non-critical parameters:

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}.

Note: SCG is a batch learning method, so shuffling the patterns has no effect.

9.17.4 Complexity of SCG

The number of epochs is not relevant when comparing SCG to other algorithms like standard backpropagation. Indeed one iteration in SCG needs the computation of two gradients, and one call to the error function, while one iteration in standard backpropagation needs the computation of one gradient and one call to the error function. Møller defines a *complexity unit* (cu) to be equivalent to the complexity of one forward passing of all patterns in the training set. Then computing the error costs 1 cu while computing the gradient can be estimated to cost 3 cu. According to Møller's metric, one iteration of SCG is as complex as around 7/4 iterations of standard backpropagation.

[13] As it is not rare that SCG can not reduce the error during a few consecutive epochs, this criterion is computed only when $E(w_{k+1}) \leq E(w_k)$. Without such a precaution, this criterion would stop SCG too early.

Note: As the SNNS implementation of SCG is not very well optimized, the CPU time is not necessarily a good comparison criterion.

Chapter 10

Pruning Algorithms

This chapter describes the four pruning functions which are available in SNNS. The first section of this chapter introduces the common ideas of pruning functions, the second takes a closer look at the theory of the implemented algorithms and the last part gives guidance for the use of the methods. Detailed description can be found in [Bie94] (for "non-contributing units") and [Sch94] (for the rest).

10.1 Background of Pruning Algorithms

Pruning algorithms try to make neural networks smaller by pruning unnecessary links or units, for different reasons:

- It is possible to find a fitting architecture this way.
- The cost of a net can be reduced (think of runtime, memory and cost for hardware implementation).
- The generalization[1] can (but need not) be improved.
- Unnecessary input units can be pruned in order to give evidence of the relevance of input values.

Pruning algorithms can be rated according to two criterions:

- *What* will be pruned? We distinguish weight pruning and node pruning. Special types of node pruning are input pruning and hidden unit pruning.
- *How* will be pruned? The most common possibilities are *penalty term algorithms* (like Backpropagation with Weight Decay, see section 9.1.4) and *sensitivity algorithms* which are described in this chapter.

Sensitivity algorithms perform training and pruning of a neural net alternately, according to the algorithm in figure 10.1.

[1] Generalization: ability of a neural net to recognize unseen patterns (test set) after training

1. Choose a reasonable network architecture.
2. Train the net with backpropagation or any similar learning function into a minimum of the network.
3. Compute the *saliency* (relevance for the performance of the network) of each element (link or unit respectively).
4. Prune the element with the smallest saliency.
5. Retrain the net (into a minimum again).
6. If the net error is not too big, repeat the procedure from step 3 on.
7. (optional) Recreate the last pruned element in order to achieve a small net error again.

Figure 10.1: Algorithm for sensitivity algorithms

10.2 Theory of the implemented algorithms

There are different approaches to determine the saliency of an element in the net. This section introduces the implemented sensitivity pruning algorithms.

10.2.1 Magnitude Based Pruning

This is the simplest weight pruning algorithm. After each training, the link with the smallest weight is removed. Thus the saliency of a link is just the absolute size of its weight. Though this method is very simple, it rarely yields worse results than the more sophisticated algorithms.

10.2.2 Optimal Brain Damage

Optimal Brain Damage (OBD) approximates the change of the error function when pruning a certain weight. A Taylor series is used for the approximation:

$$\begin{aligned} \delta E &= E(W + \delta W) - E(W) \\ &= \frac{\partial E(W)}{\partial W} \cdot \delta W + \frac{1}{2} \frac{\partial^2 E(W)}{\partial W^2} + R_2(W + \delta W) \end{aligned} \tag{10.1}$$

To simplify the computation, we assume that

- the net error function was driven into a minimum by training, so that the first term on the right side of equation (10.1) can be omitted;
- the net error function is locally quadratic, so that the last term in the equation can be left out;
- the remaining second derivative (*Hesse-matrix)* consists only of diagonal elements, which affects the second term in equation (10.1).

The result of all these simplifications reads as follows:

$$\delta E \;=\; \frac{1}{2} \sum_{1 \le i,j \le n_n} h_{(ij),(ij)} \cdot \delta w_{ij}^2 \tag{10.2}$$

Now it is necessary to compute the diagonal elements of the Hesse-Matrix. For the description of this and to obtain further information read [YLC90].

10.2.3 Optimal Brain Surgeon

Optimal Brain Surgeon (OBS, see [BH93]) was a further development of OBD. It computes the full Hesse-Matrix iteratively, which leads to a more exact approximation of the error function:

$$\delta E \;=\; \frac{1}{2}\, \delta W^T \cdot H \cdot \delta W \tag{10.3}$$

From equation (10.3), we form a minimization problem with the additional condition, that at least one weight must be set to zero:

$$\min_{(ij)} \left\{ \min_{\delta W} \left(\frac{1}{2}\, \delta W^T \cdot H \cdot \delta W \right) \;|\; E_{(ij)}^T \cdot \delta W \;+\; w_{ij} \;=\; 0 \right\} \tag{10.4}$$

and deduce a Lagrangian from that:

$$L \;=\; \frac{1}{2}\, \delta W^T \cdot H \cdot \delta W \;+\; \lambda \cdot (E_{(ij)}^T \cdot \delta W \;+\; w_{(ij)}) \tag{10.5}$$

where λ is an Lagrangian multiplier. This leads to

$$\delta W \;=\; -\frac{w_{ij}}{[H^{-1}]_{(ij),(ij)}} \cdot H^{-1} \cdot E_{(ij)} \tag{10.6}$$

$$L_{(ij)} \;=\; s_{ij} \;=\; \frac{1}{2} \frac{w_{(ij)}^2}{[H^{-1}]_{(ij),(ij)}} \tag{10.7}$$

Note that the weights of all links are updated.

The problem is, that the inverse of the Hesse-Matrix has to be computed to deduce saliency and weight change for every link. A sophisticated algorithm has been developed, but it is still very slow and takes much memory, so that you will get in trouble for bigger problems.

10.2.4 Skeletonization

Skeletonization ([MM89]) prunes units by estimating the change of the error function when the unit is removed (like OBS and OBD do for links). For each node, the *attentional strength* is introduced which leads to a different formula for the net input:

$$net_j \;=\; \sum_i w_{ij} \cdot \alpha_i \cdot o_i \tag{10.8}$$

Figure 10.2 illustrates the use of the attentional strength.

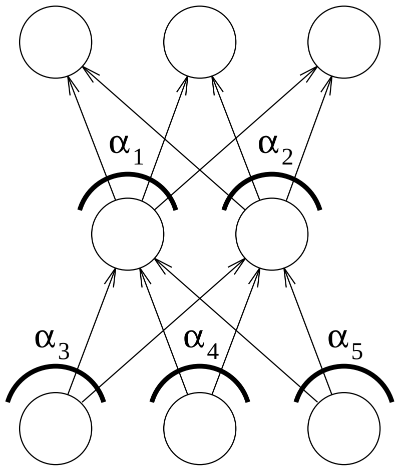

Figure 10.2: Neural network with attentional strength for each input and hidden neuron

Defining the relevance of a unit as the change in the error function while removing the unit we get

$$\rho_i \;=\; E_{\alpha_i=0} \;-\; E_{\alpha_i=1} \approx \left.\frac{\partial E}{\partial \alpha_i}\right|_{\alpha_i=1} \tag{10.9}$$

In order to compute the saliency, the linear error function is used:

$$E \;=\; E^l \;=\; \sum_j |t_{pj} - o_{pj}| \tag{10.10}$$

10.2.5 Non-contributing Units

This method uses statistical means to find units that don't contribute to the net's behavior. The net is subdivided into its layers, the output of each neuron is observed for the whole pattern set and units are removed that

- don't vary their output,
- always show the same output as another unit of the same layer or
- always show the opposite output of another unit of the same layer.

This function is the first part of the method introduced by [JS91]. For further information about the implementation read [Bie94].

10.3 Pruning Nets in SNNS

To use one of the pruning algorithms mentioned above, set the learning function in the control panel (options menu, see section 4.4) to `PruningFeedForward`.

Note: This information is saved with the net. Be sure to check the learning function each time you reload the net.

The pruning algorithms can be customized by changing the parameters in the pruning panel (see figure 10.3) which can be invoked by choosing `PRUNING` in the main menu. The figure shows the default values of the parameters.

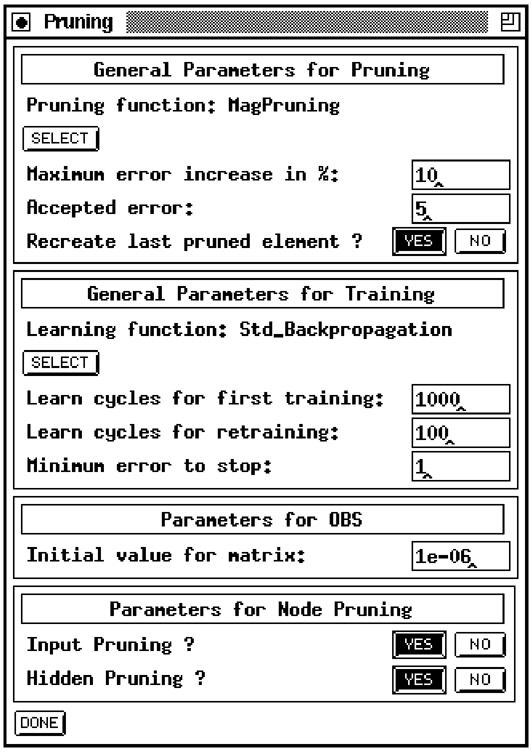

Figure 10.3: Pruning Panel

The first block of the panel controls the pruning, the second the embedded learning epochs and the last two specify parameters for certain pruning algorithms. To save the changes to the parameters and close the pruning panel, press the button `DONE`.

The current pruning function is shown in the box "`General Parameters for Pruning`". To select a different one, press the button `SELECT` which pops up a list of the available pruning functions, analogous to the change of the learning function.

There are two criterions to stop the pruning: The error after retraining must not exceed

- the error before the first pruning by more then a certain percentage determined by the user in the field "`Maximum error increase in %:`" and
- the absolute value given in the field "`Accepted error`"

Normally, the state of the net before the last (obviously too expensive) pruning is restored at the end. You can prevent this by switching the radio buttons next to "`Recreate last pruned element`" to `NO`.

The second box in the pruning panel ("`General Parameters for Learning`") allows to select the (subordinate) learning function in the same way as the pruning function. Only learning functions for feed-forward-nets appear in the list. You can select the number of epochs for the first training and each retraining separately. The training, however, stops when the absolute error falls short of the "`Minimum error to stop`". This prevents the net from overtraining.

The parameter for OBS in the third box is the initial value of the diagonal elements in the Hesse-Matrix. For the exact meaning of that parameter (to which OBS is said to be not very sensible) see [BH93].

The last box allows the user to choose which kind of neurons should be pruned by the node pruning methods. Input or hidden unit pruning can be selected by two sets of radio buttons.

Learning Parameters of the subordinate learning function have to be typed in in the control panel, as if the training would be processed by this function only. The field `CYCLES` in the control panel has no effect on pruning algorithms. To start pruning, press the button `DONE` in the pruning panel and the button `ALL` or `SINGLE` respectively in the control panel.

Chapter 11

3D-Visualization of Neural Networks

11.1 Overview of the 3D Network Visualization

This section presents a short overview over the 3D user interface. The following figures show the user interface with a simple three-layer network for the recognition of letters.

The info window is located in the upper left corner of the screen. There, the values of the units can be displayed and changed. Next to it, the 2D display is placed. This window is used to create and display the network topology. The big window below is used for messages from the kernel and the user interface. The control panel in the lower left corner controls the learning process. Above it, the 3D display is located which shows the 3D visualization of the network. The 3D control window, in the center of the screen, is used to control the 3D display.

In the upper part, the orientation of the network in space can be specified. The middle part is used for the selection of various display modes. In `SETUP` the basic settings can be selected. With `MODEL` the user can switch between solid and wire-frame model display. With `PROJECT` parallel or central projection can be chosen. `LIGHT` sets the illumination parameters, while `UNITS` lets the user select the values for visualizing the units. The display of links can be switched on with `LINKS`. `RESET` sets the network to its initial configuration. After a click to `FREEZE` the network is not updated anymore. The `DISPLAY` button opens the 3D-display window and `DONE` closes it again. In the lower part of the window, the z-coordinate for the network layers can be set.