# SNARL v1.0 Reference Manual

Simon D. Levy
Department of Computer Science
University of Tennessee
Knoxville, TN 37996-1301
levy@cs.utk.edu

January 1996

## 1 Summary

This document describes the data types and routines available in SNARL, a Simple Neural ARchitecture Library for C programmers. This document, along with SNARL source code and examples, is available by anonymous ftp from cs.utk.edu in the directory pub/levy/SNARL, and over the World Wide Web at http://www.cs.utk.edu/~levy.

I wrote SNARL because of the need on my part and others' for a small library of neural network routines providing sigma-pi units [2, 3] recurrence [1], and back-propagation in time [1]. SNARL is not a library for simple neural nets, but rather a simple library for a wide variety of neural nets: it allows you to ignore the computational support required to train test and such networks; instead, you can focus on the details of your particular application. I wrote SNARL from a dynamical-systems perspective [4], making it especially suitable for those wishing to explore the relationship between neural nets and dynamical systems.

The best way to get started with SNARL would be to download all of the material in the ftp directories, and experiment with the sample programs in the `examples` directory. I assume that the reader has some familiarity with neural networks and C programming.

I have tested most of the routines in SNARL; however, I make no claims or promises about their reliability, and I encourage users to send bug reports to levy@cs.utk.edu. I also welcome suggestions for improving or expanding the library. I have compiled and run the library on a SPARC10 running SunOS, a Silicon Graphics Inidigo running Solaris, and a VAX running VMS; it should be portable to most systems with no modification. Please send me e-mail if you encounter portability problems.

SNARL currently supports state- and parameter-dynamics; future verions of the library may support graph-dynamics as well [4]. Preliminary work has been completed on a compiler (SNARC) based on the library routines. By eliminating pointers and optimizing-out certain redundant features of the networks, the compiled version has achieved up to six-fold speedups over the equivalent library code at run-time.

# 2 Data types

SNARL provides four new data types, which can be used to declare variables in the same way that `int`, `char`, and other declarators are used in C. These new types are `Network`, `Node`, `Link`, and `Layer`. A program using SNARL may contain any number of networks, each of which may contain any number of nodes and links between nodes. The `Layer` type provides additional support for those wishing to bypass the Node level and implement the layer-based networks commonly described in the literature [2].

## 2.1 Network

At least one variable of type `Network` must be declared for each program using SNARL. By allowing any number of networks to co-exist in a given program, SNARL supports multi-network programs of the sort used in A-Life simulations. Networks are created using the `SnCreateNetwork` routine described in section 4.1 below.

## 2.2 Node

The `Node` type is the basic computational unit in SNARL. Nodes are created by one of the `SnCreateNode` functions.

Each node (a.k.a. unit, a.k.a. neuron) computes an *activation function* of its input. There are three pre-defined activation functions: Logistic, Identity, and Bias. The Logistic function is the logistic-sigmoid, or "squashing" function familiar from the neural network literature [3]:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

A node having this function can be created by the `SnCreateLogisticNode` routine. The function is typically used as the activation function for hidden and output units.

In the Identity function, the output is equal to the input; *i.e.*, $f(x) = x$. This function is typically used for input units. An identity node can be created using the `SnCreateIdentityNode` routine.

In the bias function, the output is always equal to one; *i.e.*, $f(x) = 1$. As its name implies, this function is used for bias units: Instead of having a bias as part of the activation function of a unit, you create a bias unit and connect it to that unit; the bias value is changed by modifying the weight between the two units. Nodes of this type can be created by the `SnCreateBiasNode` routine.

SNARL also allows you to create nodes with customized activation functions, using the `SnCreateNode` routine. This routine takes as arguments the name of a C routine computing the function, and the name of a C routine computing the function's first derivative (e.g., `sin` and `cos`).

## 2.3 Link

Links are the means by which nodes are connected in SNARL. Having created two nodes, you can link them via the `SnCreateLink` routine, which takes the two nodes as arguments, creates a link from the first node to the second, and returns a new identifier of type `Link`.

Each link has a *weight*, which expresses the strength of the link. These weights can be fixed at the time the link is created, or modified later by one of the routines described below.

### 2.3.1 Inputs

With a mechanism for linking two nodes, we are in a position to compute the input value to the second node, which we define as the sum of the weighted activations of the nodes linked to the node:

$$input_i = \sum_j w_{i,j} a_j \tag{2}$$

where $w_{i,j}$ is the weight on the link from node $j$ to node $i$, and $a_j$ is the activation of node $j$.

### 2.3.2 Errors

We can also define the *error* on a node, which expresses the difference between the node's actual activation and a target value based on external data. For output nodes (*i.e.*, nodes having incoming links and bound to target values via one of the `SnBind` routines), the error is equal to the difference between the target value and the actual value:

$$error_i = target_i - activation_i \tag{3}$$

For hidden nodes (*i.e.*, nodes having incoming and outgoing links), the error computation is more complicated. The error at such a node is equal to the sum of the weighted *deltas* of the nodes to which the node connects:

$$error_i = \sum_j w_{j,i}\delta_j \tag{4}$$

In this equation, $j$ indexes the nodes to which node $i$ connects; $w_{j,i}$ is the weight on the link connecting the two nodes, and $\delta_j$ is node $j$'s error mutliplied by the first derivative of its activation:

$$\delta_j = (error_j)(f'(a_j)) \tag{5}$$

It is possible for a node to have its error computed both from a target and from its outgoing connections; for example, we might wish to train hidden units based on targets placed on output units and targets placed on the hidden units themselves. In this case, SNARL adds the errors computed by both methods, and this sum becomes the error for the node.

#### 2.3.3 Delay links

SNARL supports both *delay* and *non-delay* links: In a delay link, the link's weight is multiplied by the incoming node's previous activation; in a non-delay link, the weight is multiplied by that node's current activation. Delay links are useful in the creation of *recurrent* nets; *i.e.*, nets in which activation from a node is fed back into the node itself or into some other node(s). Recurrent nets often exhibit interesting *state dynamics*, meaning that the activations of nodes change over time for a given input or initial condition [4]. Because of this capability, such nets are often trained and tested on *sequences* of input and target values, instead of single static values. SNARL supports sequences via the many binding routines described in section 4 below. Non-sequential values, such as the inputs and targets of the famed exclusive-or problem [3], are supported as special-case sequences of length one.

#### 2.3.4 Conjuncts

In addition to the additive connections provided by links, SNARL supports multiplicative connections via the *conjunct* mechanism. Having created a link

between two nodes, we can connect other nodes into this link via the `SnCojoin` routine, which takes the name of a node and the name of a link as arguments, and cojoins the node to the link.Any number of nodes can be cojoined to a link in this way; as a result, each term of the summation in equation 6 will be computed as the product of the link weight, the activation of the linked-from node, and the activations of all cojoined nodes:

$$input_i = \sum_j w_{i,j} a_j \prod_k c_{i,j,k} \tag{6}$$

where $c_{j,i,k}$ is the activation of the $k^{th}$ node cojoined to the link from node $j$ and node $i$.

In a similar way, we can rewrite equation 4 to express the effect of conjuncts:

$$error_i = \sum_j w_{j,i} \delta_j \prod_k c_{j,i,k} \tag{7}$$

# 3 Training networks in SNARL

SNARL also supports *parameter dynamics*, in which the link weights themselves change according to some algorithm [4]. In SNARL, the algorithm used to change the link weights is *back-propagation-through-time* [1], an extension of the usual back-propagation algorithm found in the literature [3]. This algorithm works as follows:

> (1) For a given pattern (pairing of input and target sequences), compute and store all node activations over the length of the sequences (often referred to as the *activation history*), by running the sequences through the net beginning-to-end. This computation is often referred to as the *forward pass*.
>
> (2) Starting at the end of each sequence and progressing toward the beginning, perform a *backward pass*, doing the following compuations at each step:
>
> > (a) Compute the node errors and deltas at that step (see section 2.3.2 above).
> >
> > (b) For each link into a node, accumulate the link's weight change as the product of the node's delta, the activation of the node at the other end of the link, and the activations of the other nodes cojoined to the link; *i.e.*,

$$\Delta w_{i,j} = \delta_i a_j \prod_k c_{i,j,k} \tag{8}$$

Weights are modified according to the following formula:

$$w_{i,j}^{t+1} = w_{i,j}^{t} + \eta \Delta w_{i,j}^{t} + \mu \Delta w_{i,j}^{t-1} \tag{9}$$

In this equation, $\eta$, the "learning-rate", determines how much of the weight change at the current training iteration $t$ is added to the weight, and $\mu$, the "momentum", determines how much of the weight change at the previous training iteration is added to the weight. [3]. (Note that training iterations are not the same as time steps.) Obviously, at least one of these coefficients must be non-zero in order for learning to take place.

In *epoch* training (also known as *batch* training), the weight change is summed over all training patterns, and then added to the weight; in *pattern* training (also known as *on-line* training), the weight change is added to the weight after each pattern is presented. SNARL supports epoch training through the `SnpStepEpoch` routine, and pattern training through the `SnpStepPattern` routine.

### 3.1 Layer

The `Layer` type provides a convenient way of implementing layer-based networks in which the "pi" part of equation 6 is not needed; that is, where there are no conjuncts. As with the Node-based routines, the Layer-based routines provide you with default activation functions (Logistic and Identity), via the `SnlCreateLogisticLayer` and `SnlCreateIdentityLayer` routines, and custom functions via the `SnlCreateLayer` routine. For further convenience, the Logistic layer has an automatic bias, so that it is not necessary to create a separate bias for the nodes in such a layer.

## 4 Routines

The names of SNARL routines all begin with `Sn`, so that they can be easily identified in your code. The routines fall into six categories, each with its own prefix: basic network creation routines (`Sn`), node-based state-dynamics routines (`Sns`), node-based parameter-dynamics routines (`Snp`), layer-based network creation routines (`Snl`), layer-based state-dynamics routines (`Snls`, and layer-based parameter-dynamics routines (`Snlp`). The remainder of this document describes each SNARL routine in detail. Routines are listed alphabetically within each category.

## 4.1 Basic network-creation routines

**SnCojoin** - cojoin a node to a link

Synopsis

```
void SnCojoin(node, link)
Node      node;
Link      link;
```

Arguments

| | |
|---|---|
| *node* | Specifies the node |
| *link* | Specifies the link |

Description

`SnCojoin` cojoins a node to a link, adding the node to the link's list of conjuncts.

**SnCreateBiasNode** - create a new bias node

### Synopsis

`Node SnCreateBiasNode`(*network*)
`Network` *network;*

### Arguments

*network* Specifies the network in which to create the node

### Description

`SnCreateBiasNode` adds a new bias node to an existing network, returning a new identifier of type `Node`. A bias node has the activation function $f(x) = 1$.

**SnCreateIdentityNode** - create a new identity node

### Synopsis

`Node SnCreateIdentityNode`(*network*)
`Network` *network;*

### Arguments

*network* Specifies the network in which to create the node

### Description

`SnCreateIdentityNode` adds a new identity node to an existing network, returning a new identifier of type `Node`. An identity node has the activation function $f(x) = x$.

**SnCreateLink** - create a new link between two nodes

### Synopsis

```
Link SnCreateLink(node_from, node_to, is_delay)
Node     node_from;
Node     node_to;
int      is_delay;
```

### Arguments

| | |
|---|---|
| *node_from* | Specifies the node to connect from |
| *node_to* | Specifies the node to connect to |
| *is_delay* | Non-zero for delay link; zero for non-delay |

### Description

`SnCreateLink` links *node_from* to *node_to*, returning an identifier of type `Link`. If *is_delay* is non-zero, the activation of *node_from* at the previous time step is used to compute the input to *node_to*; otherwise, the current activation of *node_from* is used.

**SnCreateLogisticNode** - create a new logistic-sigmoid node

Synopsis

`Node SnCreateLogisticNode`(*network*)
`Network` *network;*

Arguments

*network* Specifies the network in which to create the node

Description

`SnCreateLogisticNode` adds a new logistic-sigmoid node to an existing network, returning an identifier of type `Node`. A logistic-sigmoid node has the activation function $f(x) = 1/(1 + e^{-x})$.

**SnCreateNetwork** - create a new network

### Synopsis

```
Network SnCreateNetwork(void)
```

### Description

`SnCreateNetwork` creates a new network and returns it in an identifier of type `Network`.

**SnCreateNode** - create a new node with an arbitrary activation function

### Synopsis

```
Node SnCreateNode(network, func, dfunc)
Network    network;
double     (*func)(double);
double     (*dfunc)(double);
```

### Arguments

| | |
|---|---|
| *network* | Specifies the network in which to create the node |
| *func* | Specifies the routine that computes the activation function |
| *dfunc* | Specifies the routine that computes the first derivative of the activation function |

### Description

`SnCreateNode` adds a new customized node to an existing network, returning an identifier of type `Node`. The *func* and *dfunc* arguments refer to C routines declared earlier in the code (via a header file, *e.g.*). The user is responsible for making sure that `dfunc` accurately computes the first derivative of `func`.

## 4.2 Node-based state-dymamics routines

**SnsBindScalar** - bind a scalar to a node as a sequence

Synopsis

```
void SnsBindScalar(node, scalar, slen)
Node      node;
double    scalar;
int       slen;
```

Arguments

| | |
|---|---|
| *node* | Specifies the node |
| *scalar* | Specifies the scalar value |
| *slen* | Specifies the length of the vector created by repeating the scalar |

Description

`SnsBindScalar` converts the argument in *scalar* to a vector of length *slen* and "binds" this vector to the node specified by *node*. This is the same as calling `SnsBindVector` (*q.v.*) with all the values in the vector being equal. `SnsBindScalar` is useful when you want the acviation of a node to remain constant over all time steps in a testing sequence, as with the input node(s) of a recurrent net that outputs a time-varying sequence for a fixed input. Say, for example, that you had a network with one input node, and you wanted to test its behavior on the input

{.1, .1, .1, .1, .1, .1, .1, .1, .1, .1}.

Instead of setting up a vector of ten .1's, you could write

```
SnsBindScalar(input_node, .1, 10);
```

**SnsBindVector** - bind a vector to a node as a sequence

### Synopsis

```
void SnsBindVector(node, vector, slen)
Node      node;
double    *vector;
int       slen;
```

### Arguments

| | |
|---|---|
| *node* | Specifies the node |
| *vector* | Specifies the vector |
| *slen* | Specifies the length of the vector |

### Description

`SnsBindVector` "binds" the floating-point vector in *vector* to the node specified by *node*. In subsequent calls to `SnsStep` (*q.v.*), the successive values in the vector will be used as the activations of the node specified by *node*. `SnsBindVector` would be called on the input nodes of a net that maps one time sequence to another, when you wish to test the performance of the net.

**SnsGetActivation** - get the current activation of a node

Synopsis

```
double SnsGetActivation(node)
Node       node;
```

Arguments

*node* Specifies the node

Description

`SnsGetActivation` returns a double-precision floating-point value equal to the activation of the node *node* at the current time step.

**SnsInit** - initialize a network for testing

Synopsis

```
void SnsInit(network)
Network     network;
```

Arguments

*network* Specifies the network

Description

`SnsInit` should be called on a network each time the set of input bindings on the network changes; *i.e.*, after calling the `SnBind` routines and before calling the `SnsStep` routine.

**SnsRandomize** - randomize node activations in a network

### Synopsis

```
void SnsRandomize(network, amin, amax, seed)
Network     network;
double      amin;
double      amax;
int         seed;
```

### Arguments

| | |
|---|---|
| *network* | Specifies the network |
| *amin* | Specifies the minimum random activation value |
| *amax* | Specifies the maximum random activation value |
| *seed* | Specifies the seed for the random-number generator |

### Description

`SnsRandomize` is useful for setting up arbitrary intitial conditions when testing a network whose connection weights are known beforehand. Setting the random seed allows you to fix the quasi-random sequence used to create activations; a value of -1 for this argument causes SNARL to use the current time in seconds, since 00:00:00 GMT, January 1, 1970.

**SnsSetActivation** - set the current activation of a node

### Synopsis

```
void SnsSetActivation(node, aval)
Node      node;
double    aval;
```

### Arguments

| | |
|---|---|
| *node* | Specifies the node |
| *aval* | Specifies the activation value |

### Description

Use `SnsSetActivation` to set the current activation of a node. This routine is useful for setting up initial conditions in a network whose connection weights are known beforehand, and for "perturbing" node activations to investigate the behavior of oscillatory networks.

**SnsStep** - step a network through one iteration of testing

Synopsis

```
void SnsStep(network)
Network      network;
```

Arguments

*network* Specifies the network

Description

`SnsStep` computes the activations of all nodes in the network *network* whose activation is defined at the current testing iteration (time step, "tick"). Node activations are computed according to the following scheme: if a node has incoming links from other nodes, the activation function is computed from those links using equation 6. Otherwise, if a time-sequence of values was bound to the node via one of the `SnsBind` routines, the current value in the sequence is used. Finally, if no activation can be computed by either of these mechanisms, the algorithm checks whether an activation has been set by an earlier call to the `SnsSetActivation` routine (*q.v.*). It is unusual that more than one of these conditions would hold for a given node; however, the hierarchy is provided for those rare cases in which the activaiton could be computed in more than one way. `SnsStep` also advances the index for sequences bound to the network's nodes via one of the `SnsBind` routines. If this index excedes the network's sequence length, a non-fatal error is reported, and no computation takes place.

## 4.3 Node-based parameter-dymamics routines

**SnpBindScalar** - bind a vector of scalars to a node as a training list

### Synopsis

```
void SnpBindScalar(node, svector, nseq, slen)
Node      node;
double    *svector;
int       nseq;
int       slen;
```

### Arguments

| | |
|---|---|
| *node* | Specifies the node |
| *svector* | Specifies the vector |
| *nseq* | Specifies the number of sequences implicit in the vector |
| *slen* | Specifies the number of time steps in each sequence |

### Description

`SnpBindScalar` converts each of the *nseq* elements of the vector *svector* to a vector of length *slen* and uses the resultant vector of vectors as a "training list" for the node *node*. This is the same as calling `SnpBindVector` (*q.v.*) with all the values in each *nseq* sub-vector being equal. Sub-vectors correspond to a set of training sequences that will be used to set the activation or target of the node during training. `SnpBindScalar` is useful when you want the acviation or target to remain constant over all time steps in a training sequence, as with the input node(s) of a recurrent net that outputs a time-varying sequence for a fixed input Say, for example, that you had a network with one input node and one output node, and you wanted to train the network to map from the sequence {.1, .1, .1, .1} to some other sequence, and from the sequence {.9, .9, .9, .9} to some other sequence. Instead of setting up a vector of four .1's followed by for .9's, you could write

```
double input_vector[] = {.1, .9};

/* set up the network here */

SnpBindScalar(input_node, input_vector, 2, 4);
```

**SnpBindVector** - bind a vector of sequences to a node as a training list

### Synopsis

```
void SnpBindVector(node, vvector, dcvector, nseq, slen)
Node      node;
double    *vvector;
int       *dcvector;
int       nseq;
int       slen;
```

### Arguments

| | |
|---|---|
| *node* | Specifies the node |
| *vvector* | Specifies the vector of sequences |
| *dcvector* | Specifies a vector of don't-care flag sequences (or NULL) |
| *nseq* | Specifies the number of sequences in the vector |
| *slen* | Specifies the number of time steps in each sequence |

### Description

`SnpBindVector` "binds" the double-precision floating-point vectors in *vvector* to the node specified by *node*. Sub-vectors are assumed to be of equal length and correspond to a set of training sequences that will be used to set the activation or target sequences of the node during training. In subsequent calls to `SnpStepEpoch` or `SnpStepEpoch` (*q.v.*), the successive sub-vectors will be used to set the time-varying activations or targets of the node. `SnpBindVector` would be called on the input nodes of a net that maps one time sequence to another, when you wish to train the net on one or more input sequences. Don't-care flags are useful when you don't wish to specifiy the desired behavior of a node at all time-steps; typically, this is done to allow a network to interpolate between specified target values. A non-zero value in *dcvector* tells SNARL not to compute an error at the corresponding time step; a zero value in this vector causes the error to be computed. If no such don't-care conditions are needed, you can pass NULL for *dcvector*.
For example, consider the table on the following page, which shows a set of training data for a network with one input node and one output node, where an asterisk (*) indicates a don't-care condition:

| Pattern | Time | Input | Target |
|---|---|---|---|
| A | 1 | .1 | .8 |
| | 2 | .2 | * |
| | 3 | .3 | .6 |
| B | 1 | .6 | .3 |
| | 2 | .5 | .1 |
| | 3 | .4 | .2 |

To implement these patterns, you could write

```
double input_vector[] = {.1, .2, .3, .6, .5, .4};
double target_vector[] = {.8, 0, .6, .3, .1, .2};
int dc_vector[] = {0, 1, 0, 0, 0, 0};

/* set up the network here */

SnpBindVector(input_node, input_vector, dc_vector, 2, 3);
```

**SnpGetErrorMax** - get the current maximum error in a network

Synopsis

```
double SnpGetErrorMax(network)
Network      network;
```

Arguments

*network* Specifies the network

Description

`SnpGetErrorMax` returns a double-precision floating-point value equal to the maximum error at any time step on any output node in the network *network*. An output node is defined as a node having at least one input from another node as well as a target set by one of the `SnpBind` routines. `SnpGetErrorMax` runs a forward pass on all training patterns set up through the `SnpBind` routines, and computes the errors on all nodes at all time steps, using the method described in Section 2.3.2 above. `SnpGetErrorMax` is useful for finding the "worst-case" error in a network, whereas `SnpGetErrorRMS` (*q.v.*) gives an *average* idea of the error over all patterns and time steps.

**SnpGetErrorRMS** - get the current RMS error in a network

Synopsis

```
double SnpGetErrorRMS(network)
Network      network;
```

Arguments

*network* Specifies the network

Description

`SnpGetErrorRMS` returns a double-precision floating-point value equal to the root-mean-squared error over all output nodes and all time steps in the network *network*. An output node is defined as a node having having at least one input from another node as well as a target set by one of the `SnpBind` routines. `SnpGetErrorRMS` runs a forward pass on all training patterns set up through the `SnpBind` routines, and computes the errors on all nodes at all time steps, using the method described in section 2.3.2 above. `SnpGetErrorRMS` is useful for getting an *average* idea of the error in a network over all patterns and time steps, whereas `SnpGetErrorMax` (*q.v.*) provides a "worst-case" measurement of the error.

RMS error is computed according to the following equation, in which $n$ indexes output nodes, $p$ indexes patterns, and $t$ indexes time steps:

$$Error = \sqrt{\frac{\sum_{n=1}^{N}\sum_{p=1}^{P}\sum_{t=1}^{T}(target_{n,p,t} - output_{n,p,t})^2}{NPT}} \qquad (10)$$

**SnpGetWeight** - get the current weight on a link

Synopsis

```
double SnpGetWeight(link)
Link       link;
```

Arguments

*link* Specifies the link

Description

`SnpGetWeight` returns a double-precision floating-point value equal to the weight on the link *link* at the current training iteration.

**SnpInit** - initialize a network for training

Synopsis

```
void SnpInit(network)
Network      network;
```

Arguments

*network* Specifies the network

Description

`SnpInit` should be called on a network each time the set of pattern bindings on the network changes; *i.e.*, after calling the `SnpBind` routines and before calling the`SnpStepEpoch` or `SnpStepPattern` routine..

**SnpLoadWeights** - load network weights from a file

Synopsis

```
void SnpLoadWeights(network, file)
Network     network;
char        *file;
```

Arguments

| | |
|---|---|
| *network* | Specifies the network |
| *file* | Specifies the file name |

Description

Use `SnpLoadWeights` to restore network weights saved from an earlier training session. The *network* argument should refer to a network that has already been created exactly like the network from which the weights were saved. The *file* argument should refer to a file created by `SnpSaveWeights` (*q.v.*). Fixed weights are not loaded.

**SnpRanodmize** - randomize link weights in a network

### Synopsis

```
void SnpRanodmizeWeights(network, wmin, wmax, seed)
Network     network;
double      wmin;
double      wmax;
int         seed;
```

### Arguments

| | |
|---|---|
| *network* | Specifies the network |
| *wmin* | Specifies the minimum random weight value |
| *wmax* | Specifies the maximum random weight value |
| *seed* | Specifies the seed for the random-number generator |

### Description

`SnpRanodmizeWeights` is useful for setting up arbitrary intitial conditions when training a network. Certain values of *wmin* and *wmax* will typically cause a given network to get stuck in a local minimum; if this happens, try changing the values and re-compiling your program. Setting the random seed allows you to fix the quasi-random sequence used to create activations; a value of -1 for this argument causes SNARL to use the current time in seconds, since 00:00:00 GMT, January 1, 1970.

**SnpSaveWeights** - save network weights to a file

### Synopsis

```
void SnpSaveWeights(network, file)
Network     network;
char        *file;
```

### Arguments

| | |
|---|---|
| *network* | Specifies the network |
| *file* | Specifies the file name |

### Description

Use `SnpSaveWeights` to save network weights after training, for later restoration by `SnpLoadWeights` (*q.v.*). Weights are saved in text format, one weight per line. Fixed weights are not saved.

**SnpSetEtaMuLink** - set a link's learning rate and momentum

### Synopsis

```
void SnpSetEtaMuLink(link, eta, mu)
Link     link;
double   eta;
double   mu;
```

### Arguments

| | |
|---|---|
| *link* | Specifies the Link |
| *eta* | Specifies the learning rate |
| *mu* | Specifies the momentum |

### Description

`SnpSetEtaMuLink` should be called on each link in a network when you want different links to have different learning rates and momenta. Otherwise, you can call `SnpSetEtaMuNet` (*q.v.*) on the entire network. One of these two routines must be called in order for learning to take place.

**SnpSetEtaMuNet** - set a learning rate and momentum for all links in a network

### Synopsis

```
void SnpSetEtaMuLink(network, eta, mu)
Network     link;
double      eta;
double      mu;
```

### Arguments

| | |
|---|---|
| *network* | Specifies the network |
| *eta* | Specifies the learning rate |
| *mu* | Specifies the momentum |

### Description

`SnpSetEtaMuNet` should be called on a network when you want all links to have the same learning rate and momentum. Otherwise, you can call `SnpSetEtaMuLink` (*q.v.*) on each link. One of these two routines must be called in order for learning to take place.

**SnpSetWeight** - set a link's weight

### Synopsis

```
void SnpSetWeight(link, weight)
Link      link;
double    weight;
```

### Arguments

*link* Specifies the link
*weight* Specifies the desired weight

### Description

`SnpSetWeight` should be called when you want to set a link's weight to some previously determined value, such as a weight obtained from an earlier training session.

**SnpStepEpoch** - step a network through one iteration of training on all patterns

### Synopsis

`void SnpStepEpoch(`*network*`)`
`Network` *network;*

### Arguments

*network* Specifies the network

### Description

`SnpStepEpoch` performs the back-propagation-in-time algorithm on the specified network, computing node errors as described in section 2.3.2, and modifying link weights as described in section 3. In `SnpStepEpoch`, juncture weights are modified only after all patterns have been tested; contrast this with `SnpStepPattern` (*q.v.*).

**SnpStepPattern** - step a network through one iteration of training on a single pattern

Synopsis

`void SnpStepPattern`(*network, pattern*)
`Network` *network;*
`int` *pattern;*

Arguments

| | |
|---|---|
| *network* | Specifies the network |
| *pattern* | Specifies the pattern number (first = 0) |

Description

`SnpStepPattern` performs the back-propagation in time algorithm on the specified network, computing node errors as described in section 2.3.2, but only for the specified pattern, and modifying the link weights as described in section 3. In `SnpStepPattern`, juncture weights are modified "on-line" after the presentation of a single pattern; contrast this with `SnpStepEpoch` (*q.v.*).

## 4.4 Layer-based network-creation routines

**SnlConnectFull** - fully connect two layers

Synopsis

```
void SnlConnectFull(layer_from, layer_to, is_delay)
Layer     layer_from;
Layer     layer_to;
int       is_delay;
```

Arguments

| | |
|---|---|
| *layer_from* | Specifies the layer to connect from |
| *layer_to* | Specifies the layer to connect to |
| *is_delay* | Non-zero for delay connection; zero for no delay |

Description

`SnlConnectFull` fully connects two layers; *i.e.*, it connects every node in *layer_from* to every node in *layer_to*. If *is_delay* is non-zero, the connections are implemented as delays (input from previous activations of *layer_from*); otherwise, the connections are implemented normally (inputs from current activations of *layer_from*).

**SnlConnectFullFixed** - fully connect two layers using a single fixed weight

### Synopsis

```
void SnlConnectFullFixed(layer_from, layer_to, is_delay, weight)
Layer    layer_from;
Layer    layer_to;
int      is_delay;
double   weight;
```

### Arguments

| | |
|---|---|
| *layer_from* | Specifies the layer to connect from |
| *layer_to* | Specifies the layer to connect to |
| *is_delay* | Non-zero for delay connection; zero for no delay |
| *weight* | Specifies the fixed weight |

### Description

`SnlConnectFullFixed` is the same as `SnlConnectFull` (*q.v.*), with the addition of a parameter for fixing the weight on all connections. This routine may not be very useful and was included for the sake of completeness.

**SnlConnectOneToOne** - connect two layers in a 1:1 manner

### Synopsis

```
void SnlConnectOneToOne(layer_from, layer_to, is_delay)
Layer     layer_from;
Layer     layer_to;
int       is_delay;
```

### Arguments

| | |
|---|---|
| *layer_from* | Specifies the layer to connect from |
| *layer_to* | Specifies the layer to connect to |
| *is_delay* | Non-zero for delay connection; zero for no delay |

### Description

`SnlConnectOneToOne` connects the first node in *layer_from* to the first node in *layer_to*, the second node in *layer_from* to the second node in *layer_to*, etc. If the layers have a different number of nodes, a non-fatal error is reported, and the routine has no effect. If *is_delay* is non-zero, the connections are are implemented as delays (input from previous activations of *layer_from*); otherwise, the connections are implemented normally (inputs from current activations of *layer_from*).

**SnlConnectOneToOneFixed** - connect two layers in a 1:1 manner using a single fixed weight

### Synopsis

```
void SnlConnectOneToOneFixed(layer_from, layer_to, is_delay, weight)
Layer     layer_from;
Layer     layer_to;
int       is_delay;
double    weight;
```

### Arguments

| | |
|---|---|
| *layer_from* | Specifies the layer to connect from |
| *layer_to* | Specifies the layer to connect to |
| *is_delay* | Non-zero for delay connection; zero for no delay |
| *weight* | Specifies the fixed weight |

### Description

`SnlConnectOneToOneFixed` is the same as `SnlConnectOneToOne` (*q.v.*), with the addtion of a parameter for fixing the weight on all connections. This routine is especially useful for creating the feedback connections of a recurrent net, in which it is often not desired to modify the connection weights dynamically.

**SnlCreateIdentityLayer** - create a layer of identity nodes in a network

### Synopsis

```
Layer SnlCreateIdentityLayer(network, count)
Network     network;
int         count;
```

### Arguments

| | |
|---|---|
| *network* | Specifies the network |
| *count* | Specifies the number of nodes to create in the layer |

### Description

`SnlCreateIdentityLayer` adds a new layer of identity nodes to an existing network, returning a new identifier of type `Layer`. An identity node has the activation function $f(x) = x$.

**SnlCreateLayer** - create a layer of nodes with an arbitrary activation function in a network

### Synopsis

```
Layer SnlCreateLayer(network, count, func, dfunc)
Network     network;
int         count;
double      (*func)(double);
double      (*dfunc)(double);
```

### Arguments

| | |
|---|---|
| *network* | Specifies the network |
| *count* | Specifies the number of nodes to create in the layer |
| *func* | Specifies the routine that computes the activation function |
| *dfunc* | Specifies the routine that computes the first derivative of the activation function |

### Description

`SnlCreateLayer` adds a new customized layer to an existing network, returning an identifier of type `Layer`. The *func* and *dfunc* arguments refer to C routines declared earlier in the code (via a header file, *e.g.*). The user is responsible for making sure that `dfunc` accurately computes the first derivative of `func`.

**SnlCreateLogisticLayer** - create a layer of nodes with the logistic-sigmoid activation function in a network

### Synopsis

```
Layer SnlCreateLogisticLayer(network, count)
Network     network;
int         count;
```

### Arguments

| | |
|---|---|
| *network* | Specifies the network |
| *count* | Specifies the number of nodes to create in the layer |

### Description

`SnlCreateLogisticLayer` adds a new layer of logistic-sigmoid nodes to an existing network, returning a new identifier of type `Layer`. A logistic-sigmoid node has the activation function $f(x) = 1/(1 + e^{-x})$. `SnlCreateLogisticLayer` automatically adds a bias to each node it creates. This routine is especially useful for creating hidden and output layers.

## 4.5 Layer-based state-dynamics routines

**SnlsBindScalar** - bind a vector of scalars to a layer as a sequence

Synopsis

```
void SnlsBindScalar(layer, svector, slen)
Layer     layer;
double    *svector;
int       slen;
```

Arguments

| | |
|---|---|
| *layer* | Specifies the layer |
| *svector* | Specifies the vector of scalar values |
| *slen* | Specifies the length of the vectors created by repeating the scalars |

Description

`SnlsBindScalar` converts the argument in *svector* to a vector of vectors, each of length *slen* and "binds" these vectors to the nodes in the layer specified by *layer*. This is the same as calling `SnlsBindVector` (*q.v.*) with all the values in each sub-vector being equal. In other words, `SnlsBindScalar` is the layered version of `SnsBindScalar` (*q.v.*). Say, for example, you had trained a sequential net having two nodes in its input layer, and you wanted to test the net with the sequence {.1, .1, .1} on the first node and {.9, .9, .9} on the second node. You could write:

```
double input_vector[] = {.1, .9};

/* set up the network here */

SnlsBindScalar(input_layer, input_vector, 3);
```

**SnlsBindVector** - bind a vector of sequences to a layer

### Synopsis

```
void SnlsBindVector(layer, vvector, slen)
Layer     layer;
double    *vvector;
int       slen;
```

### Arguments

| | |
|---|---|
| *layer* | Specifies the layer |
| *vvector* | Specifies the vector |
| *slen* | Specifies the length of each sub-vector sequence |

### Description

`SnlsBindVector` treats *vvector* as if it contained a sequence of sub-vectors, each of length *slen*, and binds each sub-vector to a node in *layer*. This is useful testing the behavior of a layered net on a novel input. Say, for example, you had trained a sequential net having two nodes in its input layer, and you wanted to test the net with the sequence {.1, .2, .7} on the first node and {.6, .4, .5} on the second node. You could write:

```
double input_vector[] {.1, .2, .7, .6, .4, .5};

/* set up the network here */

SnlsBindVector(input_layer, input_vector, 3);
```

**SnlsGetActivations** - get current activations of nodes in a layer

Synopsis

```
void SnlsGetActivations(layer, vector)
Layer     layer;
double    *vector;
```

Arguments

| | |
|---|---|
| *layer* | Specifies the layer |
| *vector* | Specifies the vector in which activations will be returned |

Description

`SnlsGetActivations` returns the current activations from the nodes in *layer* in the vector *vector*, which must be big enough to hold a number of values equal to the number of nodes in *layer*. This routine is useful for testing the output of a layered network based on input set by one of the `SnlsBind` or `SnlsSetActviations` routines.

**SnlsSetActivationsScalar** - set current activations of nodes in a layer to a single value

Synopsis

```
void SnlsSetActivationsScalar(layer, scalar)
Layer     layer;
double    scalar;
```

Arguments

| | |
|---|---|
| *layer* | Specifies the layer |
| *scalar* | Specifies the value |

Description

`SnlsSetActivationsScalar` sets the current activations of all nodes in *layer* to the value *scalar*. This is the same as calling `SnlsSetActivationsVector` (*q.v.*) with all the values in the vector being the same.

**SnlsSetActivationsVector** - set current activations of nodes in a layer to values in a vector

Synopsis

```
void SnlsSetActivationsVector(layer, vector)
Layer      layer;
double     *vector;
```

Arguments

| | |
|---|---|
| *layer* | Specifies the layer |
| *vector* | Specifies the vector |

Description

`SnlsSetActivationsVector` sets the current activations of the nodes in *layer* to the values in *vector*, which should contain the same number of values as there are nodes in the layer.

## 4.6 Layer-based parameter-dynamics routines

**SnlpBindScalar** - bind a vector of scalars to a layer as training lists

### Synopsis

```
void SnlpBindScalar(layer, svector, nseq, slen)
Layer     layer;
double    *svector;
int       nseq;
int       slen;
```

### Arguments

| | |
|---|---|
| *layer* | Specifies the layer |
| *svector* | Specifies the vector of scalars |
| *nseq* | Specifies the number of sequences implicit in the vector |
| *slen* | Specifies the length of the sequences in time steps |

### Description

`SnlpBindScalar` converts each of the *nseq* elements of the vector *svector* to a vector of length *slen* and uses the resultant vector of vectors as a set of "training lists" for the nodes in layer *layer*. This is the same as calling `SnlpBindVector` (*q.v.*) with all the values in each of the *nseq* sub-vectors being equal. Sub-vectors correspond to a set of training sequences that will be used to set the activations or targets of the nodes during training. The sub-vectors should be initialized according to a pattern/node hierarchy. `SnpBindScalar` is useful when you want the acviation or target to remain constant over all time steps in a training sequence, as with the input node(s) of a recurrent net that outputs a time-varying sequence for a fixed input.

Say, for example, that you had a network with a two-node input layer and a two-node output layer, and you wanted the network to learn the following mapping:

| Pattern | Time | Input1 | Input2 | Target1 | Target2 |
|---|---|---|---|---|---|
| A | 1 | .1 | .9 | .3 | .4 |
| | 2 | .1 | .9 | .3 | .4 |
| | 3 | .1 | .9 | .3 | .4 |
| B | 1 | .8 | .2 | .5 | .6 |
| | 2 | .8 | .2 | .5 | .6 |
| | 3 | .8 | .2 | .5 | .6 |

You could set up the patterns above with the following code:

```
double input_vector[] = {.1, .9, .8, .2 };
double target_vector[] = {.3, .4, .5, .6 };


/* set up the network here */

SnlpBindScalar(input_layer, input_vector, 2, 3);
SnlpBindScalar(target_layer, target_vector, 2, 3);
```

**SnlpBindVector** - bind a vector of values to a layer as training lists

Synopsis

```
void SnlpBindVector(layer, vvector, dcvector, nseq, slen)
Layer     layer;
double    *vvector;
int       *dcvector;
int       nseq;
int       slen;
```

Arguments

| | |
|---|---|
| *layer* | Specifies the layer |
| *vector* | Specifies the vector |
| *dcvector* | Specifies a vector of don't-care flag sequences (or NULL) |
| *nseq* | Specifies the number of patterns implicit in the vector |
| *slen* | Specifies the length of the sequences in time steps |

Description

`SnlpBindVector` uses the vector *vvector* as a set of "training list" sub-vectors for the nodes in layer *layer*. Sub-vectors correspond to a set of training sequences that will be used to set the activations or targets of the nodes during training. The sub-vectors should be initialized according to a pattern/node hierarchy. Don't-care flags are useful when you don't wish to specifiy the desired behavior of a node at all time-steps; typically, this is done to allow a network to interpolate between specified target values. A non-zero value in *dcvector* tells SNARL not to compute an error at the corresponding time step; a zero value in this vector causes the error to be computed. If no such don't-care conditions are needed, you can pass NULL for *dcvector*.

Say, for example, that you had a network with a two-node input layer and a two-node output layer, and you wanted the network to learn the mapping below, where an asterisk (*) indicates a don't-care condition:

| Pattern | Time | Input1 | Input2 | Target1 | Target2 |
|---|---|---|---|---|---|
| A | 1 | .11 | .91 | * | .41 |
| | 2 | .12 | .92 | .32 | * |
| | 3 | .13 | .93 | .33 | .43 |
| B | 1 | .81 | .21 | * | .61 |
| | 2 | .82 | .22 | .52 | .62 |
| | 3 | .83 | .23 | .53 | * |

You could set up the patterns above with the following code:

```
double input_vector[] = {
.11, .12, .13, /* pattern A, input 1 */
.91, .92, .93, /* pattern A, input 2 */
.81, .82, .83, /* pattern B, input 1 */
.21, .22, .23  /* pattern B, input 2 */
};

double target_vector[] = {
0, .32, .33, /* pattern A, target 1 */
.41, 0, .43, /* pattern A, target 2 */
0, .52, .53, /* pattern B, target 1 */
.61, .62, 0  /* pattern B, target 2 */
};

int dc_vector[] = {
1, 0, 0, /* pattern A, target 1 */
0, 1, 0, /* pattern A, target 2 */
1, 0, 0, /* pattern B, target 1 */
0, 0, 1  /* pattern B, target 2 */
};

/* set up the network here */

SnlpBindVector(input_layer, input_vector, NULL, 2, 3);
SnlpBindVector(target_layer, target_vector, dc_vector, 2, 3);
```

## References


[1] M. Caudill and C. Butler. (1992) *Understanding Neural Networks: Computer Explorations*. Chapter 5: Recurrent Networks. Cambridge, MA: MIT Press.

[2] D.E. Rumelhart, G.E. Hinton, and J.L. McClelland (1986) A General Framework for Parallel Distributed Processing. In D.E. Rumelhart and J.L. McClelland, eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations*. Cambridge, MA: MIT Press.

[3] D.E. Rumelhart, G.E. Hinton, and R.J. Williams (1986) Learning Internal Representations by Error Propagation. In Rumelhart and McClelland, *op. cit.*

[4] E.L. Saltzman and K.G. Munhall (1992) Skill Acquisition and Development: The Role of State-, Parameter-, and Graph-Dynamics. *Journal of Motor Behavior*, Vol. 24, No. 1, 49-57.