3 Modeling in NSL

In chapter 2 we introduced model simulation in NSL. The models overviewed were "canned" ready for simulation, having preset parameters as well as visualization specifications. In this chapter we overview how to build neural network models in NSL using the NSLM modeling language. Note that this material is intended for the model builder, as distinct from the model user. We first explain how models are described in terms of *modules* and *neural networks* in NSLM, followed by an introduction to the *Schematic Capture System* (*SCS*), our visual tool to create and browse *model architectures*. We then describe the NSL implementation of the *Maximum Selector*, *Hopfield* and *Backpropagation* models introduced in chapter 2.

3.1 Implementing Model Architectures with NSLM

A neural network model is described by a *model architecture* representing its structure and behavior. In NSL, model architectures can be built either top-down or bottom-up. If built top-down, the two step approach to building the model is: first build *modules* to define the overall "black-box" structure of the network and then build the detailed functionality of the *neural networks*. To build bottom-up, we just do the reverse. We illustrate the bottom-up approach with the *Maximum Selector*, *Hopfield*, and *Backpropagation* models.[1]

Modules and Models

At the highest-level model architectures are described in terms of *modules* and *interconnections*. We describe in this section these concepts as well as the *model*, representing the *main* module in the architecture together with a short overview of scheduling and buffering involved with modules.

Modules

The *module*, the basic component in a model architecture, is somewhat analogous to the *object* in object-oriented applications. Additionally, the corresponding module definition is analogous to an object definition, known as the object *class*, used to *instantiate* the actual modules or objects, respectively. A module encapsulates the internal complexity of its implementation by separating the internal details from the external interface. The external portion of the module is the part of the module seen by other modules. The internal portion is not seen by other modules—this makes it easier to create and modify modules independently from each other—and defines the actual module behavior. This behavior need not be reducible to a neural network: (a) it may be an abstraction equivalent to that of a neural network, or (b) it may be a module doing something else, e.g. providing inputs or monitoring behavior.

The most important task of a module's external interface is to permit communication between different modules. As such, a module in NSL includes a set of input and output *data ports* (we shall call them simply *ports*). The port represents an entry or exit point where data may be sent or received to or from other modules, as shown in figure 3.1.

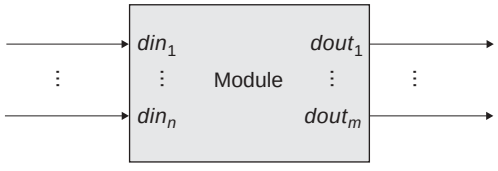

Figure 3.1
The NSL computational model is based on the module concept. Each Module consists of multiple input, din_1,din_n, and output, $dout_1,...,dout_m$, *data ports* for unidirectional communication. The number of input ports does not have to be the same as the number of output ports.

For example, the *Maximum Selector* model architecture incorporates a module having two input ports *sin* and *vin* together with a single output port *uf*, as shown in figure 3.2.

Figure 3.2
The ULayer module of the *Maximum Selector* model has two input ports *sin* and *vin* and a single output port *uf*.

Data sent and received through ports is usually in the form of numerical values. These values may be of different numerical types while varying in dimension. In the simplest form a numerical type may correspond to a single scalar, a one-dimensional array (*vector*), a two-dimensional array (*matrix*), or higher dimensional arrays.[2] For example in the *Ulayer* module shown in figure 3.2, *vin* is made to correspond to a scalar type while *sin* and *uf* both correspond to vector arrays (the reason for this selection will become clear very soon).

In terms of implementation, the NSL *module* specification has been made as similar as possible to a *class* specification in object-oriented languages such as Java and C++ in order to make the learning curve as short as possible for those already having programming background. The general module definition is described in code segment 3.1. The module specification consists of a header and a body. The header comprises the first line of the template, in other words the code outside the curly brackets. The body of the module is specified inside the curly brackets made up of the *structure* and *behavior*, both to be explained later on.

```
nslModule Name (arguments)
{
   structure
   behavior
}
```

Code Segment 3.1
The NslModule definition.

Let us begin with the header (**bold** letters represent NSLM keywords):

- **nslModule** (note the initial lower case "nsl" prefix) specifies the beginning of a module template.
- *Name* (note the initial upper case letter) represents the name of the module to which all module instances will refer.
- *arguments* are an optional variable list useful when passing external information to the module during an instantiation.
- The body of the module consists of two different sections:
- *structure* representing module attributes (data).
- *behavior* representing module methods (operations).

For example, the **Ulayer** module in the *Maximum Selector* model architecture contains the header described in code segment 3.2.

```
nslModule Ulayer (int size)
{
}
```

Code Segment 3.2
MaxSelector Ulayer header.

The header specification consists of:

- nslModule, the always present module definition keyword.
- Ulayer, the name of the module.

- size, an integer type passed as argument to the module.

The module *structure* consists of the module's external interface—its ports, as shown in code segment 3.3.

```
nslModule Ulayer(int size)
{
    public NslDinDouble1 sin(size);
    public NslDinDouble0 vin();
    public NslDoutDouble1 uf(size);
}
```

Code Segment 3.3
MaxSelector's Ulayer external interface.

The **Ulayer** module defines the three ports previously mentioned, *sin*, *vin* and *uf*. Each line ending in a semicolon defines a single port declaration:

- public tells NSLM that the port (or any other specification) is known outside the module—it is part of the module's external interface. (Defining all ports as public is very important if we want to be able to make connections or communication channels with other modules.)
- NslDinDouble1 represents a one-dimensional port array of type "double," where Nsl is the prefix to all NSL defined types. As part of the type description, Din specifies an input data port. Double specifies the primitive data type for the port (other primitive types are Float and Int) while 1 identifies the array dimension, in this case 1, for a vector (other dimensions are 0, 1, 2, 3, or 4).
- sin is the port name used for NSLM referral both from inside the module as well as from its outside.
- The parentheses after sin indicate the instantiation parameter section. In this example the parameter size in the header is passed to the module during its instantiation.
- Ports vin and uf are defined in a similar way. Port vin is of NslDinDouble0 type corresponding to an input port of zero dimensions (i.e., a scalar). uf is of NslDoutDouble1 type corresponding to a one dimensional output port array.

Besides the external interface in the form of ports, the structure of a module may include additional local data. In our example we include three additional "internal" variables *up*, *hu*, and *tau* as shown in code segment 3.4.

```
nslModule Ulayer(int size)
{
    public NslDinDouble1 sin(size);
    public NslDinDouble0 vin();
    public NslDoutDouble1 uf(size);

    private NslDouble1 up(size);
    private NslDouble0 hu();
    private double tau();
}
```

Code Segment 3.4
MaxSelector's Ulayer attribute definition.

up represents an internal module variable of type NslDouble1. Since all attributes, with the exception of ports, should be encapsulated we use the private visibility keyword to specify a local variable not viewed externally to the module. Note how the Din/Dout section of the port types is taken out from a regular variable declaration. The other section, primitive type and dimension are still important, in this case Double and 1, respectively.

- hu and tau represent the offset and approximation method time constant, both of type NslDouble0.

In terms of *behavior*, every module must have methods in order to do something "meaningful." Modules include a number of specific methods called by the simulator during model execution. These methods are used for different purposes, e.g. initialization, execution, termination.

```
nslModule Ulayer(int size)
{
    public NslDinDouble1 sin(size);
    public NslDinDouble0 vin();
    public NslDoutDouble1 uf(size);

    private NslDouble1 up(size);
    private NslDouble0 hu();
    private NslDouble0 tau();
    public void initRun() {
        uf = 0.0
        up = 0.0;
        hu=0.1;
        tau=1.0;
    }
}
```

Code Segment 3.5
MaxSelector's Ulayer attribute and method definition.

For example, the **initRun** method in the **Ulayer** module definition shown in code segment 3.5 is called during the module's run reinitialization. (Additional methods will be defined for this module later in this chapter.) Tasks that we may want to do during reinitialization are for example resetting of all variables to their initial value. (Note that we usually set values for local variables and output ports but not input ports since their values are externally received.) Every method is distinguished by its unique *signature*, consisting of a return type, name and arguments, as well as additional modifiers such as the visibility keyword. In our example the method is defined as follows:

- public is the visibility modifier telling NSLM that the method is to be known outside the module, an important requisite if we want NSL to be able to call this method during module simulation.
- void is the return type from the method, i.e., no value is returned from the method. This is the case with most NSL predefined methods.
- initRun is the name of the method, taken from the set of predefined NSL method names.

Arguments are specified within the parenthesis. In this example no arguments are passed to the method, the case with most NSL predefined methods.

The method body corresponds to the section between curly brackets. Note that the **initRun** defined here sets both the values of arrays **uf** and **up** to 0.0, in other words it assigns zero to every element in the corresponding arrays. On the other hand **hu** and **tau** are initialized to 0.1 and 1.0 respectively.

Interconnections

Interconnections between modules is achieved by interconnecting output ports in one module to input ports in another module. Interconnections free the user from having to specify how data should be sent and received during simulation processing. Communication is unidirectional, flowing from an output port to an input port. Code segment 3.6

shows the **Vlayer** header and structure (we omit its behavior) for the *Maximum Selector* model. It contain s an output **vf**, input port **uin** and three private variables, **vp**, **hv** and **tau**.

```
nslModule Vlayer(int size)
{
    public NslDinDouble1 uin(size);
    public NslDoutDouble0 vf();

    private NslDouble0 vp();
    private NslDouble0 hv();
    private NslDouble0 tau();
}
```

Code Segment 3.6
MaxSelector's Vlayer attribute definition.

The description is very similar to **Ulayer**. The major difference is that **Ulayer**'s output, **uf**, is a vector while **Vlayer**'s output, **vf**, corresponds to a single scalar. figure 3.3 then shows the interconnections between the **Ulayer** module and another, **Vlayer**.

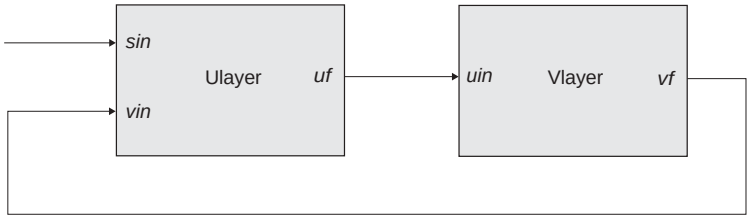

Figure 3.3
Interconnections between modules Ulayer and Vlayer of the *Maximum Selector* model.

In the example, a connection is made from output port **uf** in **Ulayer** to input port **vin** in **Vlayer**; additionally, output port **vf** in **Vlayer** is connected to input port **vin** in **Ulayer**. Note the input port **sin** in **Ulayer** is disconnected at the time. In general, a single output port may be connected to any number of input ports, whereas the opposite is not allowed, i.e., connecting multiple output ports to a single input port. The reason for this restriction is that the input port could receive more than one communication at any given time, resulting in inconsistencies.

This kind of interconnection—output to input port—is considered "same level module connectivity." The alternative to this is known as "different level module connectivity." In this case, an output port from a module at one level is *relabeled* (we use this term instead of *connected* for semantic reasons) to an output port of a module at a different level. Alternatively, an input port at one level module may be *relabeled* to an input port at a different level. For example, in figure 3.4 we introduce the **MaxSelector** module, containing an input port **in** and an output port **out**, encapsulating modules **Ulayer** and **Vlayer**. **MaxSelector** is considered a higher level module to the other two since it contains—and instantiates—them. In general, relabeling lets input and output ports *forward* their data between module levels. (This supports module encapsulation in the sense that a module connected to **MaxSelector** should not connect to ports in either **Ulayer** or **Vlayer** nor be able to get direct access to any of the modules private variables.) Relabelings, similar to connections, are unidirectional, where an input port from one module may be relabeled to a number of input ports at a different level.

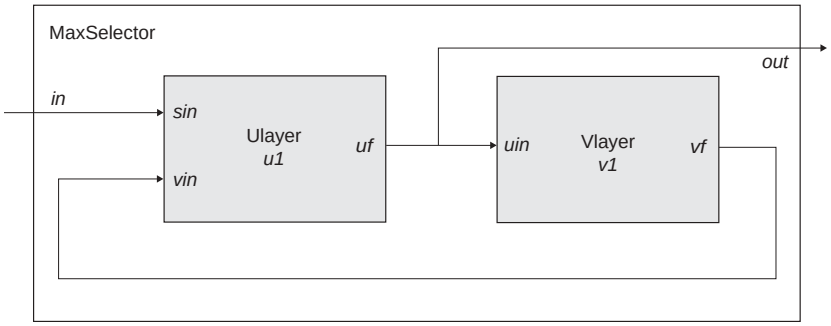

Figure 3.4
Maximum Selector model architecture contains a MaxSelector module with two interconnected modules Ulayer and Vlayer.

The NSLM specification for figure 3.4 is given in code segment 3.7. the **MaxSelector** module definition incorporates **Ulayer** and **Vlayer** instantiations—**u1** and **v1** are the corresponding instance variables—together with port **in** and **out** instantiations. Note that we have made the instantiations of Ulayer and Vlayer **private** variables. Again this is for encapsulation, or in other words, to protect these module instances from being modified accidentally.

```
nslModule MaxSelector (int size)
{
    public NslDinDouble1 in(size);
    public NslDoutDouble1 out(size);
    private Ulayer u1(size);
    private Vlayer v1(size);

    public void makeConn(){
        nslRelabel(in,u1.sin);
        nslConnect(v1.vf,u1.vin);
        nslConnect(u1.uf,v1.uin);
        nslRelabel(u1.uf,out);
    }
}
```

Code Segment 3.7
MaxSelector top-level module definition.

In terms of behavior, the **MaxSelector** module includes the predefined **makeConn** method, analogous to the **initRun** method, for specifying port interconnections. (Note that module interconnections are carried out in the *parent*—higher level—**MaxSelector** module, with **Ulayer** and **Vlayer** considered the *children*—lower level—modules.) Connections and relabels between ports are specified as follows:

- nslConnect connects an output port (first argument) to an input port (second argument). In this example we connect output port vf in v1 to input port vin in u1. The second connect statement connects output port uf in u1 to input port uin in v1.
- nslRelabel relabels an input port at a higher module level with an input port at a lower module level, or changing the order, an output port at a lower level with an output port at a higher level. In the example, we relabel input port in in MaxSelector to input port sin belonging to u1 and output port uf belonging to u1 to output port out in MaxSelector.

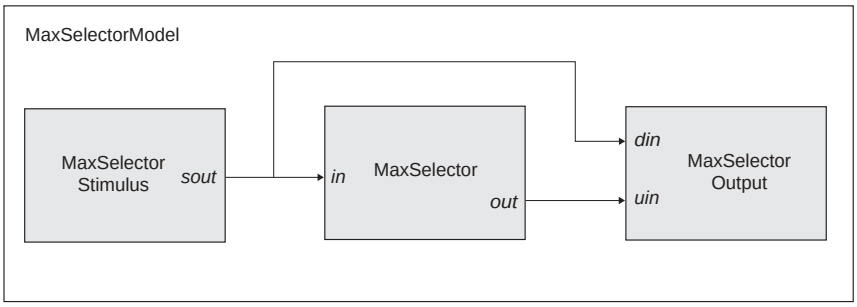

Figure 3.5
MaxSelectorModel architecture contains the MaxSelectorStimulus, MaxSelector, and MaxSelectorOutput interconnected module.

Note in the previous examples that specifying connections and relabels is carried out from outside the participating modules—the modules having the actual port to be connected. This way we can design modules independently and without priori knowledge of how they are going to be interconnected promoting module reuse applying them in a number of model architectures.

Models

There is a special module, known as the *model*, which should be present in any model architecture. The *model* is somewhat analogous to a *main* procedure in programming languages in that it is responsible for instantiating the rest of the application. The model contains the complete set of modules defining the particular model architecture. For example, the **MaxSelectorModel** is shown in figure 3.5, which includes two additional modules, **MaxSelectorStimulus** responsible for generating model input, **MaxSelectorOutput**, for the processing of module output. These two additional modules may be quite simple, as the case in our example, or may provide sophisticated functionality as probing **MaxSelector** module correctness, or make use of the output as part of further processing. The **MaxSelectorModel** definition is described in code segment 3.8.

```
nslModel MaxSelectorModel ()
{
    private MaxSelector maxselector(10);
    private MaxSelectorStimulus stimulus(10);
    private MaxSelectorOutput output();

    public void makeConn() {
        nslConnect(stimulus.sout,maxselector.in);
        nslConnect(stimulus.sout,output.sin);
        nslConnect(maxselector.out, output.uin);
    }
}
```

Code Segment 3.8
MaxSelectorModel model.

The header specification is similar to the module with the exception that we use the **nslModel** keyword instead of **nslModule**. The model may define both attributes and behavior as part of its body similar to a module. Note how we specify the network sizes and pass them to the two modules. The only restriction is that there may not be more than one model in the application. The *Maximum Selector* model is further described in the *Maximum Selector* section in this chapter. However, before we can describe the complete model, we need to discuss additional issues such as scheduling of modules and buffering of data.

Scheduling and Buffering

Before we get into the detailed implementation of modules we should understand two special aspects in processing models: *scheduling* and *buffering*. Scheduling specifies the

order in which modules and their corresponding methods are executed while buffering specifies how often ports read and write data in and out of the module. NSL uses a multi-clock-scheduling algorithm where each module clock may have a different time step although synchronizing between modules during similar time steps. During each cycle, NSL executes the corresponding simulation methods implemented by the user. We will expand upon this later in the chapter and give complete details in the NSLM chapter.

In NSL, buffering relates to the way output ports handle communication. Since models may simulate concurrency, such as with neural network processing, we have provided *immediate* (no buffer) and *buffered* port modes. In the immediate mode (sequential simulation), output ports immediately send their data to the connecting modules. In the buffered mode (pseudo-concurrent simulation), output ports do not send their data to the connecting modules until the following clock cycle. In buffered mode, output ports are double buffered. One buffer contains the data that can be seen by the connecting modules during the current clock cycle, while the other buffer contains the data being currently generated that will only be seen by the connected modules during the next clock cycle. By default NSL uses the non-buffered mode, although the user may change this. Most of the models presented in the book make use of the immediate buffering mode. Full details on scheduling and buffering are given in the NSLM chapter.

Neural Networks

As discussed in the Introduction, NSL favors model architectures where modules are implemented by neural networks. A module defines the structure and behavior of the neural network. The neural network structure consists of a set of neurons and their interconnections, whereas the neural network behavior is defined in terms of non-linear dynamics with connection weights subject to a number of learning rules.

Neurons

Without precluding the importance of other neural models, we focus here on the leaky integrator neuron model. As we described in chapter 1, Neural Networks section, the leaky integrator's internal state is described by its membrane potential or neural activity *mp* and by its output or firing *mf*, specified by some nonlinear function, as shown in figure 3.6 (drawn again from figure 1.6).

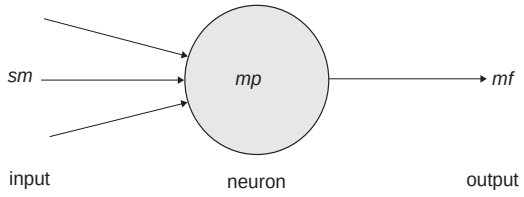

Figure 3.6
Single compartment neural model represented by a value *mp* corresponding to its membrane potential, and a value *mf* corresponding to its firing, the only output from the neuron. *sm* represents the set of inputs to the neuron.

In NSL two data structures are required to represent such a neuron in addition to its inputs. One data structure corresponds to the membrane potential and the other one to its firing rate. Different NSLM data types may be used for these structures, for example, **NslFloat0** or **NslDouble0** depending on the numerical type desired.

```
private NslDouble0 mf();
```

Notice the two variables are set to **private** with a scalar type such as **NslDouble0**. (In many cases we may want the value of *mf* to be communicated to other modules. If such is the case, the declaration for *mf* should be modified from a private variable to a public port.)

```
public NslDoutDouble0 mf();
```

In addition to the membrane potential and firing rate we need to define variable *sm* holding a weighted spatial summation of all input to the neuron

```
private NslDouble0 sm();
```

We may also need to declare the "unweighted" input to the neuron. In general, input may be specified as internal to a module or obtained from another module. In the latter case we define input *sin* as a public input port. Note that if *sin* is a vector, then *sm* is a scalar holding the sum of all values in the input vector.

```
public NslDinDouble1 sin(size);
```

The *leaky integrator* model defines the membrane potential *mp* with a first-order differential equation with dependence on its previous history and input *sm* given by equation 3.1 (combining together equations 1.1 and 1.2 and omitting the *t* parameter from both)

$$\tau \frac{dmp}{dt} = -mp + sm \tag{3.1}$$

While neural networks are continuous in their nature, their simulated state is approximated by discrete time computations. For this reason we must specify an integration or approximation method to generate as faithfully as possible the corresponding neural state. The dynamics for *mp* are described by the following statement in NSLM

```
mp=nslDiff(mp,tau,-mp+sm);
```

nslDiff defines a first-degree differential equation equal to "*-mp+ sm*" as described by the leaky integrator model. Different approximation methods can be used to approximate the differential equation. The choice of this method may affect both the computation time and its precision. For example, NSL provides Euler and Runge-Kutta II approximation methods. The selection of which method to use is specified during simulation and not as part of the model architecture. We provide further explanation on approximation methods in chapter 6.

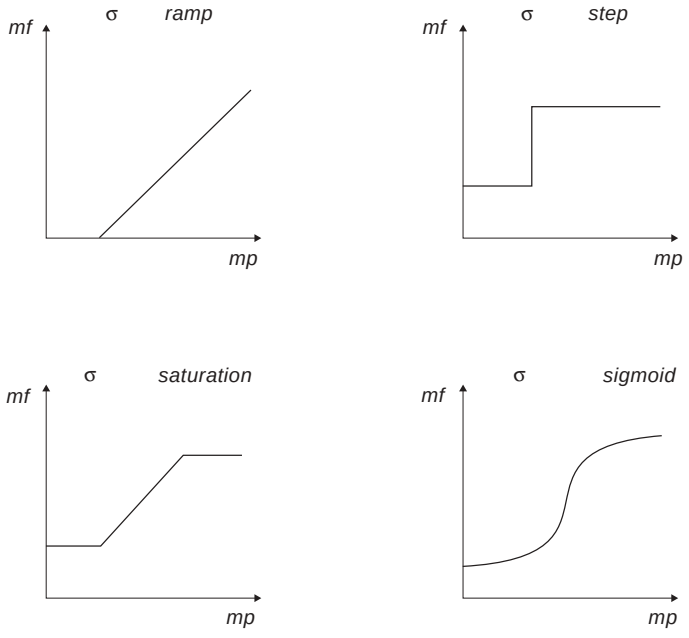

Figure 3.7
Common Threshold Functions.

The *average firing rate* or output of the neuron *mf* is obtained by applying some "threshold function" to the neuron's membrane potential as shown in equation 3.2 (taking out the *t* parameter from previous equation 1.3),

$$mf = \sigma(mp) \tag{3.2}$$

where σ usually is described by a non-linear function. For example, if σ is set to a *step* threshold function, the NSLM equation for the firing rate *mf* would be described by

```
mf = nslStep(mp);
```

where **nslStep** is the corresponding NSLM *step* threshold function. Some of the threshold functions defined in NSL are *step*, *ramp*, *saturation* and *sigmoid*, whose behaviors are plotted in figure 3.7 and described in detail in chapter 6, the NSLM language.

Neural network dynamics are generally specified inside the **simRun** method, as described code segment 3.9.

Code Segment 3.9
Leaky Integrator neuron implementation.

```
public void simRun()
{
    sm=nslSum(sin);
    mp = nslDiff(mp,tau,-mp+sm);
    mf = nslStep(mp);
}
```

While **initRun** is executed once prior to the "run," **simRun** gets executed via multiple iterations during the "run." A "run" is defined as execution over multiple clock cycles (simulation time steps) from time equal zero to the *runEndTime*. Similar to **initRun**, the **simRun** method must also be specified as **public** in order for NSL to be able to process it.

Interconnections

The previous definition specifies a single neuron without any interconnections. An actual neural network is made of a number of interconnected neurons where the output of one neuron serves as input to other neurons. In the leaky integrator neural model, interconnections are very simple structures. On the other hand, *synapses*, the links among neurons, are—in biological systems—complex electrochemical systems and may be modeled in exquisite detail. However, many models have succeeded with a very simple synaptic model: with each synapse carrying a connection weight that describes how neurons affect each other. The most common formula for the input to a neuron is given by equation 3.3 (omitting the *t* parameter from previous equation 1.4),

$$sv_j = \sum_{i=0}^{n-1} w_{ij} uf_i \tag{3.3}$$

where uf_i is the firing of neuron u_i whose output is connected to the *j*th input line of neuron v_j, and w_{ij} is the weight for that link, as shown in figure 3.8 (*up* and *vp* are analogous to *mp*, while *uf* and *vf* are analogous to *mf*).

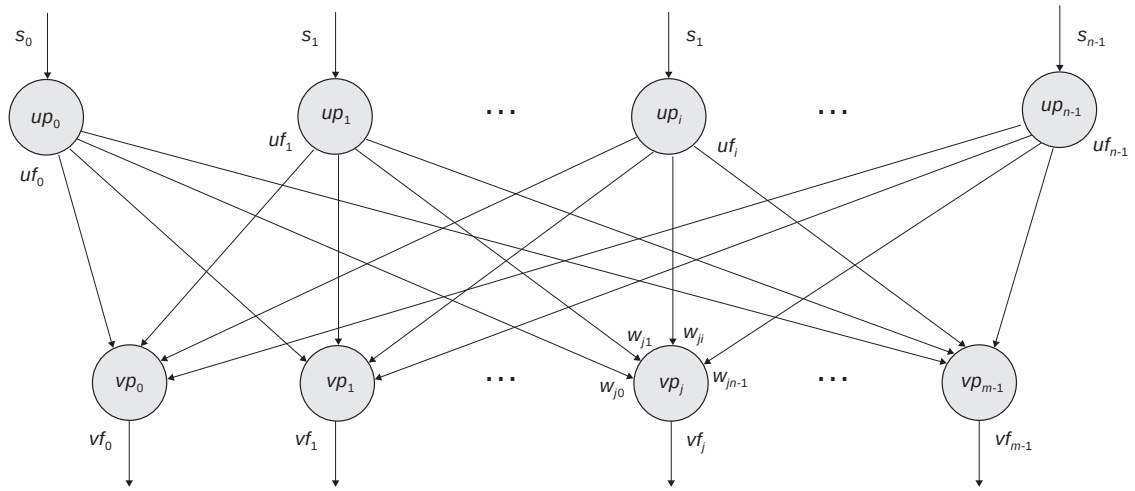

Figure 3.8
Neurons vp_j receive input from neuron firings, $uf_0,\ldots,uf_{n-1}$, multiplied by weights $w_{j0},\ldots,w_{jn-1}$, respectively.

Expanding the summation, input to neuron v_j (identified by its corresponding membrane potential vp_j) is given by sv_j defined by equation 3.4

$$sv_j = w_{0j}uf_0 + w_{1j}uf_1 + w_{2j}uf_2 + \ldots + w_{n-1,j}uf_{n-1} \tag{3.4}$$

While module interconnections are specified in NSLM via a **nslConnect** method call, doing this with neurons would in general be prohibitively expensive considering that there may be thousands or millions of neurons in a single neural network. Instead we use mathematical expressions similar to those used for their representation. For example, the input to neuron v_j, represented by *svj*, would be the sum for all outputs of neuron *ufi* multiplied (using the '*' operator) by connection weight *wij*, correspondingly.

```
svj = w0j*uf0 + w1j*uf1 + w2j*uf2 + ...;
```

Note that there exist *m* such equations in the network shown in figure 3.8. We could describe each neuron's membrane potential and firing rate individually or else we could make all u_i and v_j neuron vector structures. The first approach would be very long, inefficient, and prone to typing errors; thus we present the second approach and describe it in the following section.

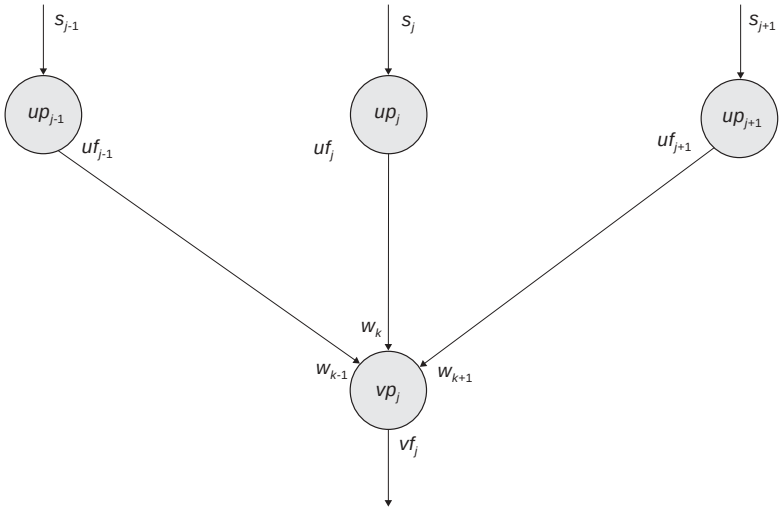

Figure 3.9
Mask connectivity.

Arrays and Masks

Instead of describing neurons and links on a one by one basis, we extend the basic neuron abstraction into *neuron arrays* and *connection masks* describing spatial arrangements among homogeneous neurons and their connections, respectively. We consider uf_j the output from a single neuron in an array of neurons and sv_j the input to a single neuron in another array of neurons. If mask w_k (for $-d \le k \le d$) represents the synaptic weights, as shown in figure 3.9, from the uf_{j+k} (for $-d \le k \le d$) elements to sv_j, for each j, we then have

$$sv_j = \sum_{k=-d}^{d} w_k uf_{j+k} \tag{3.5}$$

where the same mask w is applied to the output of each neuron uf_{j+k} to obtain input sv_j. In NSLM, the convolution operation is described by a single symbol "@".

```
sv = w@uf;
```

This kind of representation results in great conciseness, an important concern when working with a large number of interconnected neurons. Note that this is possible as long as connections are regular. Otherwise, single neurons would still need to be connected separately on a one by one basis. This also suggests that the operation is best defined when the number of *v* and *u* neurons is the same, although a non-matching number of units can be processed using a more complex notation.

To support arrays and masks, a **NslDouble1** or higher dimensional array structure is used, as was demonstrated in chapter 2, using the Hopfield model. In the Hopfield model neurons are organized as two-dimensional neuron arrays—instead of one dimensional—and weights result in four dimensional arrays—instead of two dimensional. For simplification, both neural arrays and connection masks are represented in NSLM with similar array types. The **simRun** method describing dynamics for neuron *v* would be as shown in code segment 3.10.

```
public void simRun()
{
    sv = w@uf;
    vp = nslDiff(vp,tau,-vp+sv);
    vf = nslStep(vp);
}
```

Code Segment 3.10
Leaky-Integrator simRun method implementation.

There are special considerations with convolutions regarding edge effects—a mask centered on an element at the edge of the arrays extends beyond the edge of the array—depending on how out of bound array elements are treated. The most important alternatives are to treat edges as *zero*, *wrap* around array elements such as if the array was continuous at the edges, or *replicate* boundary array elements. We will explain this in more detail in chapter 6, The NSLM Language.

3.2 Visualizing Model Architectures with SCS

There are two ways to develop a model architecture: by direct programming in NSLM as previously explained or by using the *Schematic Capture System* (*SCS*). *SCS* is a visual programming interface to NSLM that serves both as a browser as well as a tool for creating new model architectures as discussed in the Simulation chapter. While *SCS* does not provide the full programming functionality of NSLM, it provides visual support in designing modules and their interconnections. We will show in this section how to visualize already created model architectures with *SCS*. Extended details on how to created new model architectures will be overviewed in chapter 4, the *Schematic Capture System.*

To start executing the *Schematic Capture System* we invoke (see Appendix IV for platform particulars):

```
prompt> scs
```

The system initially presents the *Schematic Editor* (*SE*) window (shown in figure 3.10).

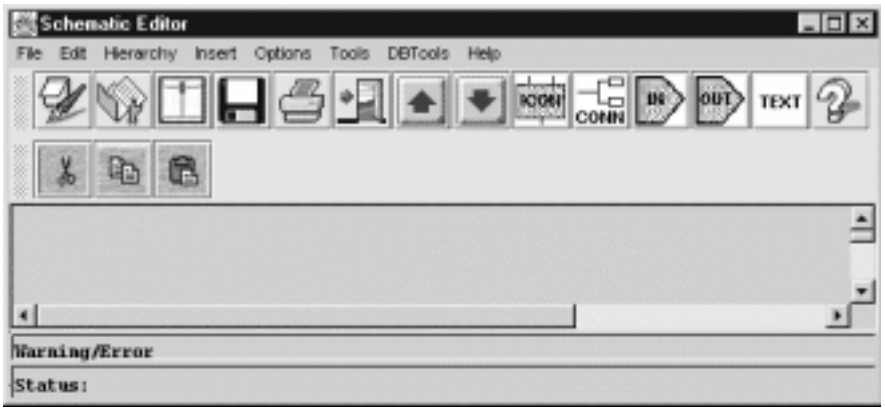

Figure 3.10
Select "Open" from the "File" menu to bring a listing of schematics available in the library of models.

To open the schematic of an existing model from the library of models we select "Open" from the "File" menu, as shown in figure 3.10. *SCS* presents a list of models, where we select for example the *MaxSelectorModel*, as shown in figure 3.11.

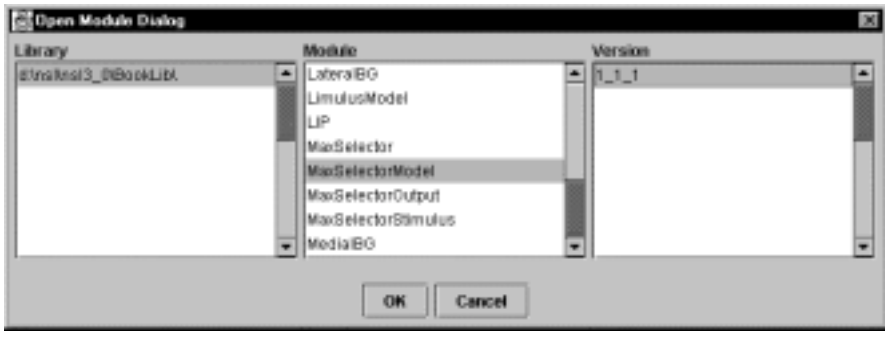

Figure 3.11
Select the MaxSelectorModel Schematic.

Once the model has been selected it is shown in the canvas section of the window, as shown in figure 3.12.

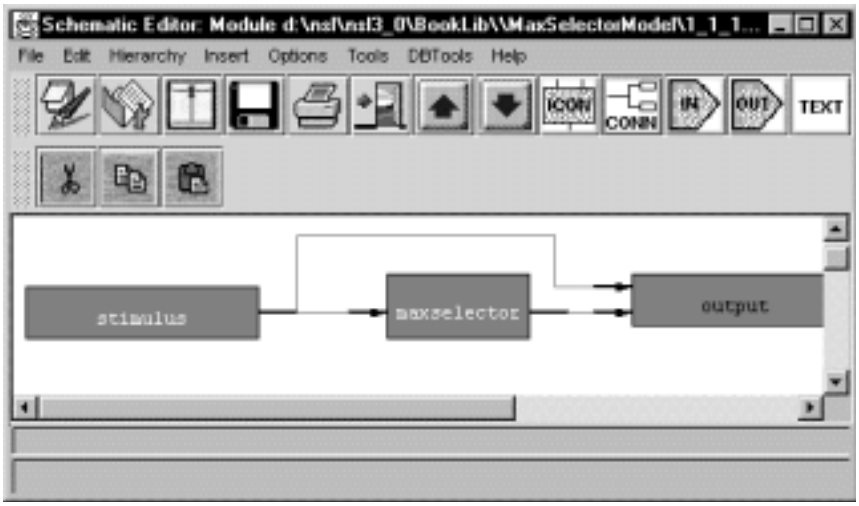

Figure 3.12
MaxSelectorModel Schematic. (The "descend" selection requires us to first select the module that we are to display.)

By “double clicking” on the **MaxSelector** module we will descend one level down the module hierarchy, and the schematics shown in figure 3.13 will be displayed.

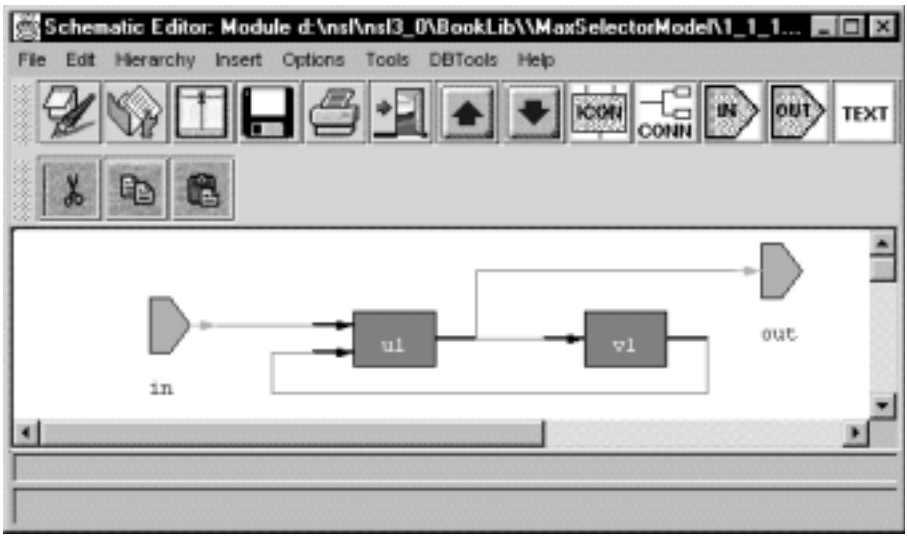

Figure 3.13
MaxSelector module from the MaxSelectorModel.

We can then return one level up by selecting “Ascend” from the “Hierarchy” menu, as shown in figure 3.14. The rest of the SCS interface is described in chapter 4.

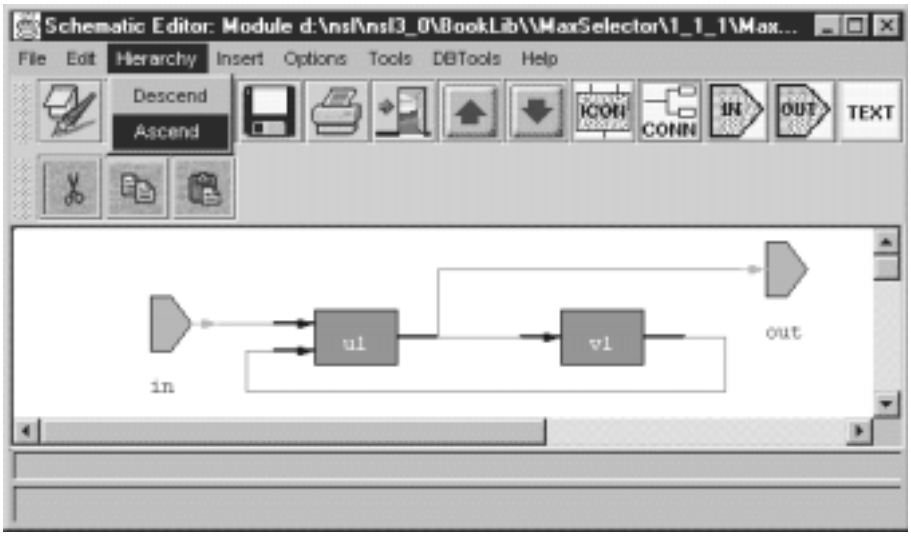

Figure 3.14
Select “ascend” from the “Hierarchy” Menu to go one level up the module hierarchy.

3.3 Maximum Selector

We presented an introduction to *Maximum Selector* model in chapter 2. We now describe its complete model architecture.

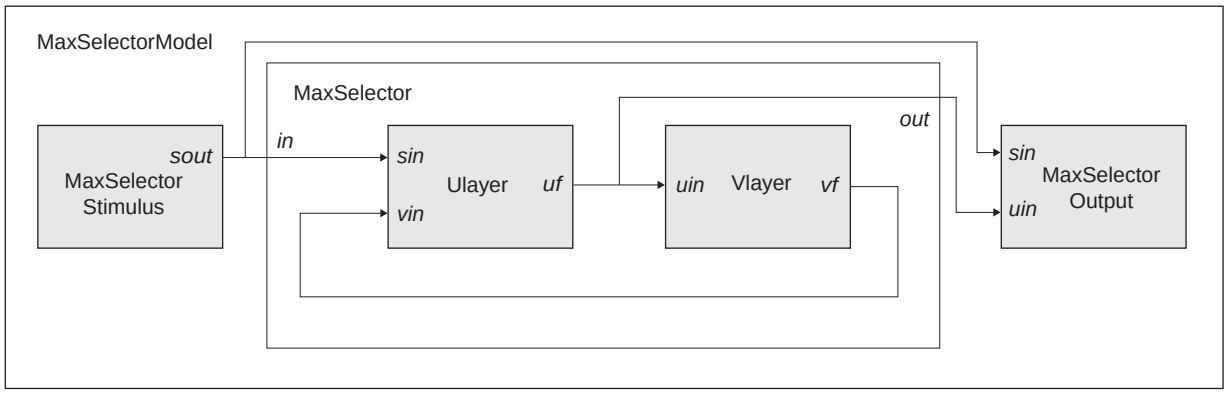

Figure 3.15
The MaxSelectorModel contains the MaxSelector module incorporating an input port in used as input to the network and an output port out represents network output, and two module instances of type Ulayer and Vlayer, respectively. Additionally, the MaxSelectorStimulus module generates stimulus for the model and MaxSelectorOutput displays the output results.

Model Implementation

As we have seen, the **MaxSelectorModel** is composed of five instances of modules: **u1, v1, maxSelector, stimulus, and output** of corresponding module types **Ulayer**, **Vlayer**, **MaxSelector**, **MaxSelectorStimulus**, and **MaxSelectorOutput**, as shown again in figure 3.15. Given an input vector *in* of array size *size*, the **maxSelector** module generates as output a similarly sized vector pattern *out* in which the only active unit (or neuron) corresponds—under suitable conditions—to the largest of the *n* vector inputs. These modules were introduced throughout the chapter so we will quickly recall and extend their description.

Ulayer

For simplicity we have kept only the minimum structure for the model (weights in this module have "1" as their value) described in code segment 3.11.

```
nslModule Ulayer (int size)
{
    public NslDinDouble1 sin(size);
    public NslDinDouble0 vin();
    public NslDoutDouble1 uf(size);
    private NslDouble1 up(size);
    private NslDouble0 hu();
    private NslDouble0 tau();
    public void initRun() {
        up =0;
        uf = 0;
        hu = 0.1;
        tau =1.0;
    }
    public void simRun() {
        up = nslDiff(up, tau, -up + uf - vin – hu + sin);
        uf = nslStep(up,0.1,0.1.0);
    }
}
```

Code Segment 3.11
Ulayer module definition.

Note that *sin*, *up*, and *uf* are vector arrays, while *vin*, *tau* and *hu* are scalar. The module behavior is described by the two equations introduced in chapter 2, with slight modifications for better correspondence with the module structure:

$$tau\frac{dup}{dt} = -up + uf - vin - hu + sin \tag{3.6}$$

$$uf = \begin{cases} 1 & \text{if } up > 0 \\ 0 & \text{if } up \le 0 \end{cases} \tag{3.7}$$

There are two separate initializations: the first, **initModule** gives script access to the *hu* offset variable; the second, **initRun** resets the neuron activity values—the values being computed during the simulation—restarting the network with a new initial state. The **simRun** method contains the above expressions to be repeatedly executed during the simulation process. Note that the two statements for *up* and *uf*, respectively, require vector array return types since both structures are vectors. When adding or subtracting a vector with a scalar, such as "uf—vin" in the **nslDiff** expression, *vin* is subtracted from every element to *uf* as if *vin* were a vector with all elements having the same value.

Vlayer

Again, for simplicity we have kept only the minimum structure for the module without the weight terms, described in code segment 3.12. Note that *uin* is a vector array, while *vp*, *vf*, *tau* and *hv* are scalars.

Code Segment 3.12
MaxSelector's Vlayer module definition.

```
nslModule Vlayer(int size)
{
    public NslDinDouble1 uin(size);
    public NslDoutDouble0 vf();
    private NslDouble0 vp();
    private NslDouble0 hv();
    private NslDouble0 tau();
    public void initRun() {
        vp =0;
        vf = 0;
        hv=0.5;
        tau=1.0;
    }
    public void simRun() {
        vp = nslDiff(vp,tau, -vp + nslSum(uin) – hv);
        vf = nslRamp(vp);
    }
}
```

The module behavior is described by the two equations introduced in chapter 2, with slight modifications for better correspondence with the module structure

$$tau\frac{dvp}{dt} = -vp + \sum_n uin - hv \tag{3.8}$$

$$vf = \begin{cases} vp & \text{if } vp > 0 \\ 0 & \text{if } vp \le 0 \end{cases} \tag{3.9}$$

These equations are implemented in the **simRun** method above. Note that the two statements for *vp* and *vf*, respectively, require this time a scalar return type since both structures are scalars. For this reason we apply **nslSum()** to all array element values from *uin*, the output received from *uf*, to obtain a single scalar value. **nslRamp** is a *ramp* threshold function.

MaxSelector

The **MaxSelector** module instantiates both **Ulayer** and **Vlayer**, as well as defining two external ports, *in* and *out*, as shown in code segment 3.13. Port interconnections are made inside the **makeConn** method, two connections and two relabels.

Code Segment 3.13
MaxSelector module.

```
nslModule MaxSelector (int size)
{
    public Ulayer u1(size);
    public Vlayer v1(size);
    public NslDinDouble1 in(size);
    public NslDoutDouble1 out(size);

    public void makeConn(){
        nslConnect(v1.vf,u1.vin);
        nslConnect(u1.uf,v1.uin);
        nslRelabel(in,u1.sin);
        nslRelabel(u1.uf,out);
    }
}
```

MaxSelectorStimulus

The **MaxSelectorStimulus** generates the visual stimulus sent to the **MaxSelector** module. The module is described in code segment 3.14. The actual stimulus can be set directly as part of the module definition, or as we discussed in the previous chapter, interactively assigned by the user through the visual interface or through the NSLS shell window. If done directly in the module definition, the **initRun** method would contain for example the corresponding stimulus specification.

Code Segment 3.14
MaxSelectorStimulus module.

```
nslModule MaxSelectorStimulus (int size)
{
    public NslDoutDouble1 sout(size);
    public void initRun(){
        sout=0;
        sout[1]=1.0;
        sout[3]=0.5;
    }
}
```

MaxSelectorOutput

The **MaxSelectorOutput** receives input from both **MaxSelectorStimulus** and **MaxSelector** modules and generates the canvases/graphs shown in the chapter 2, figure 2.8. For the sake of simplicity we leave the detailed description of this module until chapter 5, The User Interface and Graphical windows.

MaxSelectorModel

The **MaxSelectorModel** instantiates both the **MaxSelectorStimulus, MaxSelector**, and **MaxSelectorOutput** as shown again in code segment 3.15.

```
nslModel MaxSelectorModel()
{
    public MaxSelectorStimulus stimulus(size);
    public MaxSelector maxselector(size);
    public MaxSelectorOutput output(size);
    private int size = 10;

    public void initSys() {
        system.setRunEndTime(10.0);
        system.setRunStepSize(0.1);
    }
    public void makeConn(){
        nslConnect(stimulus.sout,maxselector.in);
        nslConnect(stimulus.sout,output.sin);
        nslConnect(maxselector.out,output.uf);
    }
}
```

Code Segment 3.15
MaxSelectorModel model definition.

As an exercise the user may want to add the different weight parameters specified by the original equations and change their value to see their effect on the model. Additionally, the network could be modeled with different neuron array sizes to see how this affects overall behavior.

3.4 Hopfield

Recall from the Simulation chapter the *Hopfield* model description. We now describe the model architecture.

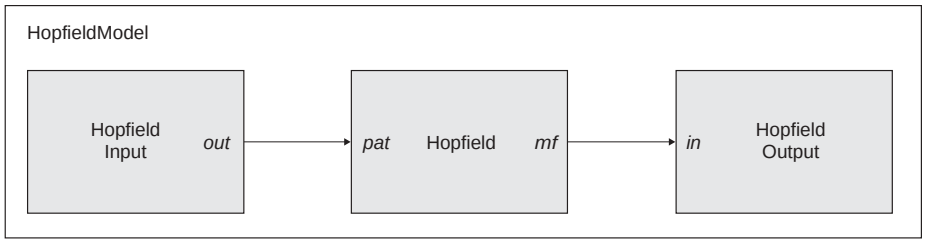

Figure 3.16
The Hopfield Model. The Hopfield module contains an input port pat used as input to the network, and an output port *mf* representing network output. The HopfieldInput module contains a single output port *out* while the HopfieldOutput module contains a single input port *in*.

Model Implementation

The *Hopfield* model contains three module instances **hopfield, in, and out** of the corresponding module types **Hopfield**, **HopfieldInput** and **HopfieldOutput** as shown in figure 3.16. Input port *pat* in **Hopfield** receives a number of initial patterns from output port *out* during training to adjust the network's connection weights. During model execution *pat* receives from *out* a single input pattern to be associated to the one that it best approximates.

Hopfield

The **Hopfield** module implements the neural network dynamics. In our example both *pat* and *mf* are set to two- dimensional arrays associating letter images to be tested (although both could have been implemented by vectors). In our example, both input and output ports have the same size for both dimensions. The structure is shown in code segment 3.16.

```
nslModule Hopfield (int size)
{
    public NslDinInt2 pat(size,size);
    public NslDoutInt2 mf(size,size);
    private NslInt2 mp(size,size);
    private NslInt2 pmf(size,size);
    private NslInt4 w(size,size,size,size);
    private NslInt0 energy();
    private NslInt0 change();
}
```

Code Segment 3.16
Hopfield module definition.

Notice that we are describing a binary *Hopfield* network, thus we implement all types as integers. Two additional two-dimensional arrays are defined, *mp* storing the activity and output of neurons and *pmf* storing previous output. We also define a four-dimensional connection matrix *w*, and two parameters, *energy* and *change*. We define a connection matrix *w* instead of a connection mask due to the varying weight values for all connections. We lose a bit on efficiency due to the higher dimension arrays but we add the ability to better map units to images. The last attribute defined is invisible to the user, but thrown in by the pre-parser. It is a primitive-type integer *size* storing the *size* passed to the module. This variable is later used in the module methods to avoid having to obtain the size each time.

During the training phase, model weights are initialized according to equation 2.7 from chapter 2 and in correspondence to our particular *Hopfield* network application at hand,

$$w_{klij} = \begin{cases} 0 & k = i, l = j \\ \sum_m pat_{mkl}\, pat_{mij} & \text{otherwise} \end{cases} \tag{3.10}$$

Besides the **initRun** and **simRun** methods, the model also requires making use of the **initTrain** and **simTrain** methods for network training. Inside the **initTrain** method, weights are set by going over all *n* input patterns and applying the above equation. Each iteration of the **initTrain** method adds a new pattern to the weight computation. Thus, the **initTrain** method has to be executed for as many times as patterns exist.

```
for (int k=0; k<size; k++)
   for (int l=0; l<size; l++)
      for (int i=0; i<size; i++)
         for (int j=0; j<size; j++)
            if (k==l && l==i && i==j)
               w[k][l][i][j]=0;
            else

w[k][l][i][j]=w[k][l][i][j]+pat[k][l]*pat[i][j];
```

Notice that NSLM enables the user to write his/her own matrix manipulation functions when necessary, such as in the above example.

After weights have been set, the network executes according to equations 2.4 and 2.5 from chapter 2

$$mp_{kl} = \sum_i \sum_j w_{klij}\, mf_{ij} \tag{3.11}$$

$$mf_{kl} = \begin{cases} 1 & \text{if } mp_{kl} \geq 0 \\ -1 & \text{if } mp_{kl} < 0 \end{cases} \tag{3.12}$$

The **initRun** method initializes neuron activity to the input pattern. One important aspect previously mentioned in the Simulation chapter is that the network gets updated asynchronously, with neurons randomly chosen for update. Thus, we compute the output for each neuron immediately after computing its activity, so it can be used as input by the next neuron chosen for update. The neuron activity *mp* and the neuron output *mf* are computed according to the above equations. The simulation proceeds as described in the **simRun** method. Since it contains a number of interesting expressions, we explain each one separately.

We first obtain two random integer values, *k* and *l*, used as indices in selecting the next neuron to be updated. Recall that all neurons are stored in a two-dimensional array and thus the need for the two indices.

```
int k = nslRandom(0,(size-1));
int l = nslRandom(0,(size-1));
```

The method **nslRandom** is one of a number of NSL library methods (described in more detail in the NSLM chapter) for numerical computations, in this case to obtain an integer random number between 0 and "size-1." Since array indices start at 0 we do not want numbers that are equal to or larger than the size of the arrays. We then apply a summation over all inputs to the randomly selected neuron multiplied by the respective connection weights—we use the "^" operator for *pointwise* array multiplication, i.e., multiplication between corresponding array elements.

```
mp[k][l] = nslSum(w[k][l]^mf);
```

Note that "w[k][l]" generates a matrix array, resulting in a valid operation. Once we get the activity for the neuron we compute its firing with a *step* function with outputs set to either -1 or 1, as specified by the last two parameters of the following **nslStep** expression. (The zero term specifies the threshold.)

```
mf[k][l] = nslStep(mp[k][l],0,-1,1);
```

We then check if the error is zero, corresponding to the output of all neurons generating exactly the same values as in the previous computation—previous values are stored in *pmf*. This is done by first subtracting *mf* from *pmf*, and then transforming the result to its absolute value to make sure all differences are positive. Finally, we add the resulting absolute values together to generate the *change*, as shown next,

```
change = nslSum(nslAbs(pmf-mf));
```

If *change* is zero, convergence has occurred, we print out a "Convergence" message—we use the **nslPrintln** printing method—and we break the simulation cycle effectively stopping execution by using the **system**.**breakCycles** method as follows

```
if (change == 0)
   nslPrintln("Convergence");
   system.breakCycles();
}
```

The module conditional ending is very common in many neural networks, especially those involving training, as will also be seen in the *Backpropagation* model.

The last expression in the **simRun** method sets the new *mf* value to the previous *pmf* value for the next simulation iteration.

```
pmf[k][l] = mf[k][l];
```

Additionally, we compute the energy in the network, described by equation 2.6 from chapter 2,

$$E = -\frac{1}{2}\sum_k \sum_l \sum_i \sum_j w_{klij} mp_{kl} mp_{ij} \tag{3.13}$$

This equation is implemented as follows

```
energy = 0;
for (int k=0; k<size; k++)
   for (int l=0; l<size; l++)
      for (int i=0; i<size; i++)
         for (int j=0; j<size; j++)
            energy=energy+w[k][l][i][j]*mp[k][l]*mp[i][j];
energy = -0.5*energy;
```

Again we need to do our own element by element multiplication.

HopfieldModel

The **HopfieldModel** instantiates the **Hopfield**, **HopfieldInput** and **HopfieldOutput** modules. Both the **HopfieldInput** and **HopfieldOutput** modules are quite simple, analogous to the MaxSelectorModel, and we leave their description until chapter 5, The User Interface and Graphical Windows. Model input can be set directly as part of the module definition or, as we discussed in the previous chapter, interactively assigned by the user via the script or menu interface. We pass the network size to the two modules, in this case "10." Module connections are made inside the **makeConn** method as described in code segment 3.17.

As an exercise the user may want to change network size as well as extend the model to make use of two different sizes for array rows and columns, respectively, instead of a single one. If you have a different problem at hand, such as the "Traveling Salesman," you may want to modify the equation for weights as well as the energy function. The rest of the computation should be the same.

```
nslModel HopfieldModel ()
{
   private int size = 10;
   public Hopfield hopfield(size);
   public HopfieldInput in(size);
   public HopfieldOutput out(size);

   public void makeConn(){
          nslConnect(in.out,hopfield.pat);
          nslConnect(hopfield.mf,out.in);
   }
}
```

Code Segment 3.17
HopefieldModel model definition.

3.5 Backpropagation

Recall from Section 2.6 the *Backpropagation* model description. We now describe the model architecture.

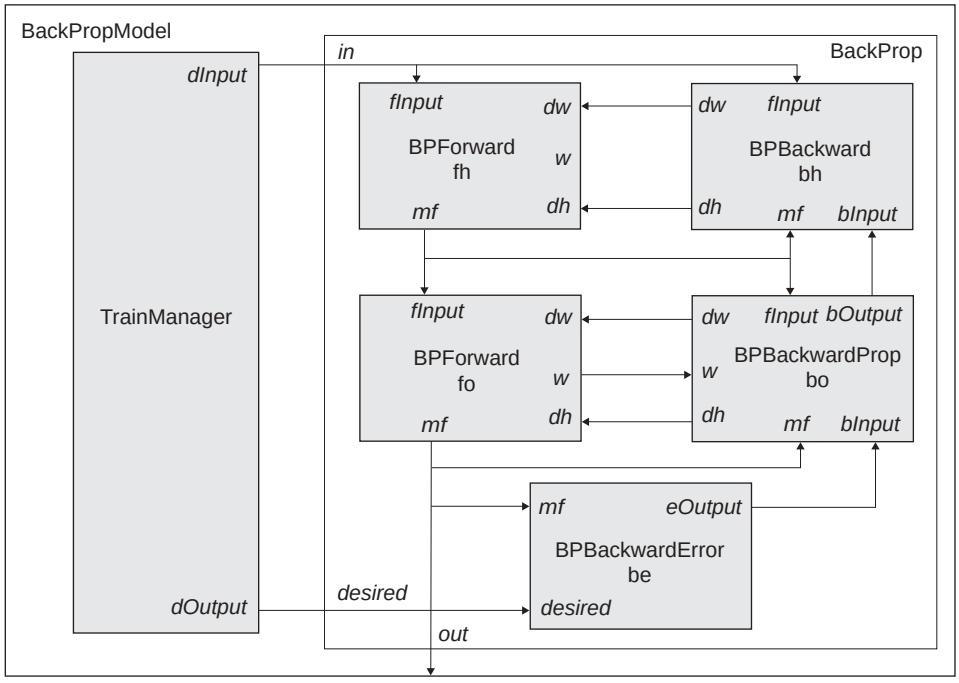

Figure 3.17
Backpropagation model architecture. The BackPropModel is decomposed into the TrainManager module where training data is read feeding training signals into the BackPropLayers module where actual neural dynamics are described. The BackPropLayers module is further decomposed into two BPForward modules, and the BPBackward, BPBackwardProp and BPBackwardError modules.

Model Implementation

The model architecture is shown in figure 3.17. The **BackPropModel** is decomposed into a **TrainManager** module and a **BackPropLayers** module. The **TrainManager** module reads training data and sends it one cycle at a time to the **BackPropLayers** module. The **BackPropLayers** module communicates with to **BPForward**, **BPBackward**, **BPBackwardProp** and **BPBackwardProp** modules to implement neural behavior. Note that there could be more modules if there were additional hidden layers in the network. Also note that we do not model the input layer as a separate module since this layer simply reads training or simulation data without any additional processing. Thus, the input layer can be directly mapped to the input port of the **BackPropLayers** module. All ports in the network are set to a one-dimensional array or vector of similar size.

The network is processed in two modes, training and simulation. Training proceeds itself in two modes:

- A feedforward mode where an input is selected and processed by the network in order to generate an output vector.
- A backpropagation mode where the error is computed as the difference between the current output vector and the desired or target output vector. The backward pass then adjusts the synaptic weights of the output layer neurons and then the hidden layer neurons, in that particular order.

As a result of this continual "wearing away" of the synaptic weights, the network will, in general, come to provide outputs that better and better approximate the target values.

Once the network has been completely trained, a process that finishes depending on the error produced simulation, the simulation run proceeds in a single mode:

A **feedforward mode** where an input vector is processed by the network to generate an output vector. The forward pass is executed a single time with the desired input pattern.

It is the job of the **BackPropLayers** module (and **BPForward**, **BPBackward**, **BPBackwardProp** and **BPBackwardProp** submodules) to conduct the forward pass and backward pass after being supplied by the **TrainManager** module with the training pair. It is the job of the **TrainManager** to take a whole training set and cycle through calling **BackPropLayers** for each pair.

We next describe the different modules top-down: **TrainManager**, **BackPropLayers**, **BPForward**, **BPBackward**, **BPBackwardProp**, **BPBackwardProp** and finally **BackPropModel**.

```
nslModule TrainManager (int nPats, int inSize, int outSize)
{
    public NslDoutFloat1 dInput(inSize);
    public NslDoutFloat1 dOutput(outSize);
    private NslFloat2 pInput(nPats, inSize);
    private NslFloat2 pOutput(nPats, outSize);
    private int counter = 0;
    private int numPats = nPats;
}
```

Code Segment 3.18
TrainManager attribute definition.

TrainManager

The **TrainManager** module reads training data from a training file and sends them one cycle at a time to the **BackPropLayers** module. The module structure is described in code segment 3.18.

The model structure includes two output ports sending both input and desired output data, *dInput* and *dOutput*, respectively. To simplify the example, we directly assign the training data into two-dimensional variables *pInput* and *pOuput*. This is quite efficient when having small training sets such as in this case. With very large data sets this would not be possible since data files may be too large to store in memory and it will be more efficient to load them from a file as needed. We include two additional parameters, *counter* and *numPats*. Variable *counter* is used to control the particular data being sent during the training cycle as will be explained next. Variable *numPats* stores the number of patterns being used for training. The actual layer sizes are passed into the module during module instantiation.

The module's main responsibility is to provide with training data to the rest of the model. Recall the training set format from table 3.1.

<num_patterns>
<input1> ... <inputN> <output1> ... <outputM>
<input1> ... <inputN> <output1> ... <outputM>
:
<input1> ... <inputN> <output1> ... <outputM>

Table 3.1
TrainManager input file format.

During model training, the **simTrain** method sends out the data. This is achieved by grabbing the next (input, desired output) pair in the given order, as shown in code segment 3.19.

```
public void simTrain()
{
    counter++;
    int pat = counter%numPats;
    dInput = pInput[pat];
    dOutput = pOutput[pat];
}
```

Code Segment 3.19
TrainManager simTrain method.

Note how we control this by doing a "mod"—operator %—of the current cycle *counter* over the number of total patterns in *numPats*. Although the training procedure is in the given order or *sequential*, where the total number of training steps is set equal to the total number of patterns, it does not have to be always that way. There are many other approaches, e.g., having the training given as a random choice of next (input, desired output) pair.

BackPropLayers

The **BackPropLayers** module is responsible for controlling the detailed training cycle. It defines data structures and simulation methods for the network necessary to execute the feedforward (activation) mode and backpropagation (adaptation) mode. The module instantiates five modules, two **BPForward** modules and single **BPBackward**, **BPBackwardProp**, and **BPBackwardError** modules. The module also defines two input ports and two output ports to receive and send information from and to the **TrainManager** module, respectively. The module definition is described in code segment 3.20.

```
nslModule BackPropLayers (int inSize, int hidSize, int outSize)
 {
    public BPForward fh(inSize,hidSize);
    public BPForward fo(hidSize,outSize);
    public BPBackwardError be(outSize);
    public BPBackwardProp bo(hidSize,outSize);
    public BPBackward bh(inSize,hidSize);

    public NslDinFloat1 in(inSize);
    public NslDinFloat1 desired(outSize);
    public NslDoutFloat1 out(outSize);
}
```

Code Segment 3.20
BackPropLayers module attributes.

Two input ports *in* and *desired* receive their data from the **TrainManager** and are relabeled to *fInput* and *desired* in the respective submodules. *fInput* represents the input layer feeding data into the hidden layer corresponding to both the first **BPForward** and the **BPBackward** modules. On the other hand *desired* is used to compute the backward error in the **BPBackwardError** module. In terms of **BackPropLayers** output ports, *out* is relabeled from the output *mf* of the second **BPForward** module representing the feedforward mode output. Note the specific order in specifying the five submodules. Since we are using immediate mode buffering this order is significant and results in the following:

1. **Process** fh BPForward **module.**
2. **Process** fo BPForward **module.**
3. **Process be BPBackwardError module.**
4. **Process** bo BPBackwardProp **module.**
5. **Process** bh BPBackward **module.**

Remember that two processing modes are involved in the model. It is crucial that the feedforward mode for both hidden and output layer be completed before the backpropagation mode on both layers. After each training *epoch*—a complete pass through all the training patterns—the network should have learned something and can use the new weights to estimate better and fine-tune itself. The network requires many epochs for training. While **BPForward** modules are processed in both modes, the last three modules are only involved in the backpropagation mode as will be seen in the rest of the section.

BPForward

The **BPForward** module implements both the hidden and output layers in the "forward" computations. The module attributes, three input ports, *fInput*, *dw* and *dh*, two output ports *mf* and *w*, and two internal variables *mp* and *h*, as shown in code segment 3.21.

```
nslModule BPForward (int inSize, int hidSize)
{
    public NslDinFloat1 fInput(inSize);
    public NslDinFloat1 dh(hidSize);
    public NslDinFloat2 dw(inSize, hidSize);
    public NslDoutFloat1 mf(hidSize);
    public NslDoutFloat2 w(inSize, hidSize);

    private NslFloat1 mp(hidSize);
    private NslFloat1 h(hidSize);
}
```

Code Segment 3.21
BPForward module attributes.

This module is involved in both feedforward and backpropagation modes. Since backpropagation mode is processed before the feedforward mode let us start describing first the module's behavior during the backpropagation mode.

Backpropagation mode

The module first initializes network thresholds and weights to random values inside the **initSys** method that gets executed every time the complete model gets reinitialized, as shown in code segment 3.22.

```
public void initSys()
{
    nslRandom(h,-1.0, 1.0);
    nslRandom(w,- 1.0, 1.0);

    dw = 0.0;
    dh = 0.0;
}
```

Code Segment 3.22
BPForward initSys method.

The **nslRandom** function sets the variables, *h* and *w*, to random values between the two limits, -0.5 and 0.5, in this case. The model also sets the two "deltas," *dw* and *dh* to 0.

We then define a "forward" computation (used in both the backpropagation and feedforward modes) calculating the activity *mp* by doing a weight matrix multiplication with over input vector *fInput*. This input is received from the previous stage, i.e. hidden

layer receives input from the input layer and the output layer receives input from the hidden layer output. The output *mf* is computed by applying a *sigmoid* threshold function over the activity. This output will then be fed into the next **BPForward** module in the case of the hidden layer, or to the **BPBackwardError** module in the case of the output layer.

The two equations (for every *i*) are as follows,

$$mp_i(t) = \sum_j w_{ji}(t) fInput_i(t) \tag{3.14}$$

$$mf_i(t) = f(mp_i(t) + h_i(t)) = \frac{1}{1 + e^{-(mp_i(t)+h_i(t))}} \tag{3.15}$$

Since we define a single **BPForward** module to define both the hidden layer and output layer "forward" dynamics, we use a single set of equations for the two, using the indices *i* and *j* instead of *s*, *p*, and *q* as originally used in Section 2.6 for the two layers. These two statements are stored in the **forward** method shown in code segment 3.23.

```
public void forward()
{
   mp = w*fInput;
   mf = nslSigmoid(mp + h);
}
```

Code Segment 3.23
BPForward forward method.

Notice that $fInput_i$ corresponds to in_s (we use *fInput* instead of the original *in* to be consistent with the rest of the modules here to distinguish between forward and backward input), while **nslSigmoid** is a NSL library function computing the *sigmoid* transfer function in the above equation.

In the backpropagation mode the thresholds and weights get updated by adding in new "deltas," *dh* and *dw* (for every *i*) computed from the previous backpropagation cycle.

$$h_i(t+1) = h_i(t) + \Delta h_i(t) \tag{3.16}$$

$$w_{ji}(t+1) = w_{ji}(t) + \Delta w_{ji}(t) \tag{3.17}$$

The backpropagation mode is stored in the **simTrain** method as described in code segment 3.24. It consists of the two updates with the "deltas" and the "forward" computation.

```
public void simTrain()
{
   w = w + dw;
   h = h + dh;
   forward();
}
```

Code Segment 3.24
BPForward simTrain method.

Feedforward mode

During the feedforward mode the **simRun** method is executed as described in code segment 3.25. It simply calls the "forward" computation.

```
public void simRun()
{
   forward();
}
```

Code Segment 3.25
BPForward simRun method.

BPBackwardError

The **BPBackwardError** module does only backpropagation mode computation. The module includes two input ports *mf* and *desired*, an output port *eOutput* and three local parameters *change*, *tss* and *pss*, as shown in code segment 3.26.

```
nslModule BPBackwardError (int outSize)
{
    public NslDinFloat1 mf(outSize);
    public NslDinFloat1 desired(outSize);
    public NslDoutFloat1 eOutput(outSize);

    private NslFloat1 stopError();
    private NslFloat0 pss();
    private NslFloat0 tss();
    public void initModule() {
          stopError.nslSetAccess('W');
    }
}
```

Code Segment 3.26
BPBackwardError module attributes.

The module receives the output *mf* from the **fo BPForward** output layer and compares its value against the desired value being forwarded from the **TrainManager** as follows

$$eOutput(t) = desired(t) - mf(t) \tag{3.18}$$

The network stops its training when a small enough error *tss* has been reached. The error calculation is as follows,

$$tss = \sum_{t} eOutput^{2}(t) \tag{3.19}$$

The computation is implemented in the **simTrain** method as shown in code segment 3.27. In order to compute the epoch error *tss* we compute first a train cycle error *pss* being accumulated through the epoch.

```
public void simTrain()
{
    eOutput = desired - mf;
    pss = pss + nslSum(eOutput ^ eOutput);
}
```

Code Segment 3.27
BPBackwardError simTrain method.

To stop training we compare the tss value against a previously set *change* value—telling the model when to stop learning—as given in the **endTrain** method, a method called at the end of every epoch completion, shown in code segment 3.28.

```
public void endTrain()
{
    tss = pss;
    if (tss < change) {
          nslPrintln("Convergence");
          system.breakEpochs();
          return;
    }
}
```

Code Segment 3.28
BPBackwardError endTrain method.

We first print a message ("Convergence") telling the user that we have completed the training cycle. This completion is actually achieved through the **system.breakEpochs** method specifying that epoch processing should be interrupted (as opposed to a **system.breakCycles** method for breaking a single training cycle). The interruption is only processed internally by NSL after the return statement.

BPBackwardProp

The **BPBackwardProp** module is only involved in the backpropagation mode. The module is defined in code segment 3.29 and it contains four input ports, *fInput*, *bInput*, *mf* and *w*, three output ports, *bOutput*, *dw* and *dh*, and two variables, *delta* and *lrate*.

```
nslModule BPBackwardProp (int hidSize, int outSize)
{
    public NslDinFloat1 bInput(outSize);
    public NslDinFloat1 fInput(hidSize);
    public NslDinFloat1 mf(outSize);
    public NslDinFloat2 w(hidSize, outSize);
    public NslDoutFloat1 dh(outSize);
    public NslDoutFloat2 dw(hidSize, outSize);
    public NslDoutFloat1 bOutput(hidSize);

    private NslFloat1 delta(outSize);
    private NslFloat0 lrate();
}
```

Code Segment 3.29
BPBackwardProp module attributes.

The module receives the **BPBackwardError** output layer error *eOuput* in *bInput* using it to compute output layer "deltas." The computation is as follows (η represents the learning rate *lrate*),

$$\delta_q(t) = mf_q(t) \times (1\text{- } mf_q(t)) \times bInput_e(t) \tag{3.20}$$

$$\Delta h_q(t) = \eta \delta_q(t) \tag{3.21}$$

$$\Delta w_{pq}(t) = \eta \delta_q(t) \times fInput_p(t) \tag{3.22}$$

$$bOutput_p(t) = \sum_q w_{qp}(t) \delta_q(t) \tag{3.23}$$

The **simTrain** method computes output the deltas, δ, *dh* and *dw*, and the output *bOutput* sent to the hidden layer as shown in code segment 3.30.

```
public void simTrain()
{
    delta = (mf * (1.0 - mf)) * bInput;
    dw = lrate * delta * fInput;
    dh = lrate * delta;
    bOutput = w*delta; //this is the product of a matrix
        times a vector
}
```

Code Segment 3.30
BPBackwardProp simTrain method.

BPBackward

The **BPBackward** module is similar to **BPBackwardProp** module except that it does not compute the additional *bOutput* (unless additional hidden layers are present). The module is defined in code segment 3.31 and it contains three input ports, *fInput*, *bInput*, and *mf*, two output ports, *dw* and *dh*, and two variables, *delta* and *lrate*.

```
nslModule BPBackward (int inSize, int hidSize)
{
    public NslDinFloat1 bInput(hidSize);
    public NslDinFloat1 fInput(inSize);
    public NslDinFloat1 mf(hidSize);
    public NslDoutFloat1 dh(hidSize);
    public NslDoutFloat2 dw(inSize, hidSize);

    private NslFloat1 delta(hidSize);
    private NslFloat0 lrate();
}
```

Code Segment 3.31
BPBackward module attributes.

The computation is as follows,

$$\delta_p(t) = mf(t) \times (1 - mf_p) \times bInput_q \tag{3.24}$$

$$\Delta h_p(t) = \eta \delta_p(t) \tag{3.25}$$

$$\Delta w_{sp}(t) = \eta \delta_p(t) \times fInput_p(t) \tag{3.26}$$

The **simTrain** method computes output the deltas, δ, *dh* and *dw*, as shown in code segment 3.32.

```
public void backwardPass()
{
    delta = mf * (1.0-mf) * bInput;
    dw = lrate * delta * fInput;
    dh = lrate * delta;
}
```

Code Segment 3.32
BPBackward simTrain method.

BackPropModel

The **BackPropModel** is responsible for instantiating its two submodules, **BackProp-Layers** and **TrainManager**, as well as initializing the appropriate layer sizes and the number of patterns to be stored. The code is shown in code segment 3.33.

```
nslModel BackPropModel ()
{
    int inSize = 2;
    int hidSize = 2;
    int outSize = 1;
    int nPats = 4;
    public TrainManager train(nPats,inSize,outSize);
    public BackPropAllLayers layers(inSize,hidSize,outSize);
}
```

Code Segment 3.33
BackPropModel model attributes.

There are a number of exercises that can be done on this model. In particular, since *Backpropagation* is a gradient descent algorithm, there are concerns that the slope of the error surface could contain local minima that the network could become stuck in. An additional training parameter known as *momentum* (identified by α) could be defined in the BPBackwardProp module. The momentum variable is quite useful in order to keep the network from becoming stuck in local minima. In such a way, the error computation equations would be modified to contain both the training rate and momentum terms as follows, for the hidden layer

$$\Delta h_p(t+1) = \eta\delta_p + \alpha \Delta h_p(t) \quad (3.27)$$

$$\Delta w_{sp}(t+1) = \eta\delta_p \times fInput_p + \alpha \Delta w_{sp}(t) \quad (3.28)$$

and for the output layer

$$\Delta h_q(t+1) = \eta\delta_q + \alpha \Delta h_q(t) \quad (3.29)$$

$$\Delta w_{pq}(t+1) = \eta\delta_q \times fInput_q + \alpha \Delta w_{pq}(t) \quad (3.30)$$

Parameter α, the momentum constant, is commonly set to around 0.9.

An additional common modification to the algorithm is to update thresholds and weights at the end of each epoch instead of every training cycle. You can modify this and see the effect on model training as well.

On a different perspective, most of the modifications on the *Backpropagation* model are usually in terms of layer sizes and input file structure. We invite the user to make the layer sizes parameters of the model instead of constant. The actual values could then be interactively assigned or read from a NSLS script file. The model would require the use of dynamic memory allocation (the **nslMemAlloc** method handling dynamic memory allocation described in the NSLM chapter).

Another possible modification is to take advantage of NSL object-oriented programming, in particular *inheritance*. Inheritance is quite useful in avoiding class definition duplication. For example, **BPBackward** and **BPBackwardProp** are quite similar. We could make **BPBackwardProp** definition inherit from **BPBackward** where we would only need to add the backward output port *bOutput* to the **BPBackwardProp** definition while the rest gets inherited. Again, we invite the user to exercise this but only after having read the NSLM chapter.

3.6 Summary

We have shown how modeling of neural architectures is done following the module approach in NSL. The three models described in this chapter, *Maximum Selector*, *Hopfield* and *Backpropagation*, use different features of NSL although keeping a very consistent organization based on the NSL module architecture.

Notes

1. The complete syntax for NSLM as well as further descriptions is found in chapter 6.
2. At the moment NSL supports up to 4-dimensional array ports and in general numerical type arrays.