2 Simulation in NSL

We will concentrate in this chapter primarily on how to run already existing models and leave new model development for the next chapter. Three neural networks simulated in NSL will be overviewed in this chapter: *Maximum Selector*, *Hopfield* and *Backpropagation*. Simulation in NSL requires a basic level of understanding of neural networks. The models chosen here will help the novice gain that understanding because of their simplicity and importance in the area of neural networks.

2.1 Selecting a Model

The simulation process begins with the selection of an already developed model; the modeling process which creates such models will be described in chapter 4, the Schematic Capture System.

However, if you do not have SCS, then to select a model from the **BookLib** models, simply change directories to where the desired model is located following the path <*installation-site*>/nsl3_0/BookLib/<modelname>. (Note that if you are working on a PC, you will want to specify the path using backward slashes "\" instead.) From there you will want to change directories to the first version, 1_1_1, and then to the **src** directory. From there either type:

```
nslj model_name
```

or

```
nslc model_name
```

These commands will invoke NSL and load the model specified. Make sure that your system administrator has set up your environment correctly. There are several environment variables we use for both NSL implemented in C++ and Java. These are discussed in chapter 5, The User Interface and Graphical Windows. See Appendix V for further details on executing models for the different platforms.

To select a model from the SCS archive of **BookLib** models, we must first open the library by calling the *Schematic Capture System* (SCS) responsible for model management (see Appendix IV for platform particulars). We execute from a shell (or by double clicking).

```
prompt> scs
```

The system initially presents the *Schematic Editor* (*SE*) window as shown in figure 2.1.

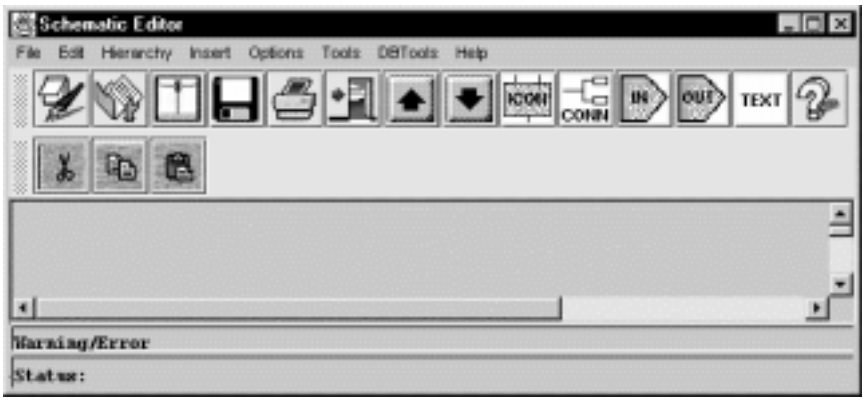

Figure 2.1
Schematic Editor Window. The different menu and button options control the creation and modification of model schematics.

To execute an existing model we select “Simulate Using Java” (or “Simulate Using C++”) from the “Tools”menu, as shown in figure 2.2.

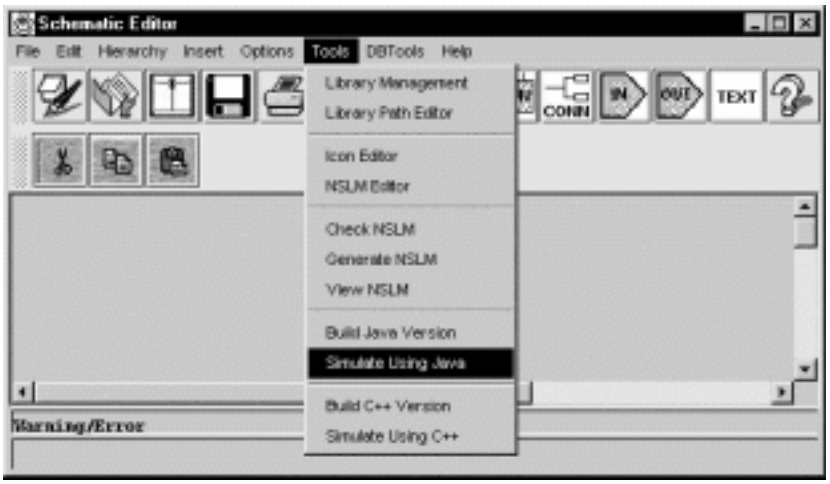

Figure 2.2
Select “Simulate Using Java” from the “Tools” menu to bring a listing of models available in the library of models and modules which are available for use in Java.

SCS then presents a list of available models, as shown in figure 2.3.

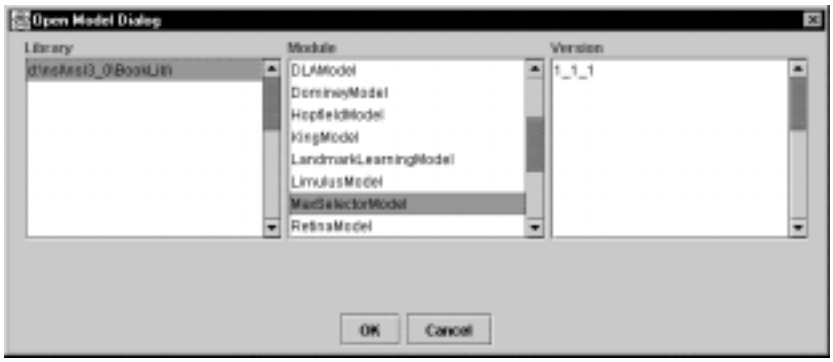

Figure 2.3
Open Model for Execution Window

For example, to choose the *MaxSelectorModel*, we select the model and version found under “nsl3_0/BookLib/ /MaxSelectorModel/1_1_1/”.

Once we chose the particular model, the system brings up the NSL Executive window presented in figure 2.4 together with an additional output display window particular to this model shown in figure 2.5. At this point we are ready to simulate the selected model. Yet, before we do that, we will quickly introduce the NSL Simulation Interface.

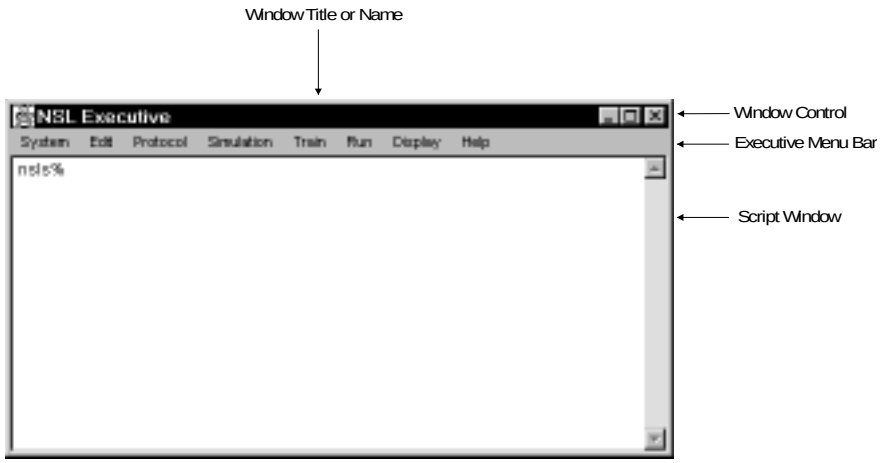

Figure 2.4
The NSL Executive window. The top part of the window contains the title and underneath the title is the Executive Menu Bar. The larger section of the window contains the NSL Script Window or shell.

2.2 Simulation Interface

The NSL **Executive** window, shown in figure 2.4, is used to control the complete simulation process such as visualization of model behavior. Control is handled either via mouse-driven menu selections or by explicitly typing textual commands in the NSL Script (NSLS) window. Since not all possible commands are available from the menus, the "NSLS" window/shell is offered for more elaborate scripts.

The top part of the window (or header) contains the window name, NSL Executive, and the Window Control (right upper corner) used for *iconizing*, enlarging and closing the window. Underneath the header immediately follows the **Executive Menu Bar**, containing the menus for controlling the different aspects involved in a simulation. The lower portion of the window contains the **Script Window**, a scrollable window used for script command entry, recording and editing. The NSL Script Language is a superset of the pull down menus in that any command that can be executed from one of the pull-down menus can also be typed in the Script window, while the opposite is not necessarily so. Furthermore, commands can also be stored in files and then loaded into the Script window at a later time. The NSLS language supports two levels of commands. The basic level allows *Tool Command Language* commands (TCL) (Ousterhout 94) while the second level allows NSL commands. The NSL commands have a special "nsl" prefix to distinguish them from TCL commands. These commands are overviewed later in the chapter and are discussed thoroughly in chapter 7, the NSL Scripting Language.

While there is a single **NSL Executive/Script** window per simulation there may be any number of additional output and input windows containing different displays. For example, the *Maximum Selector* model brings up the additional output frame shown in figure 2.5.

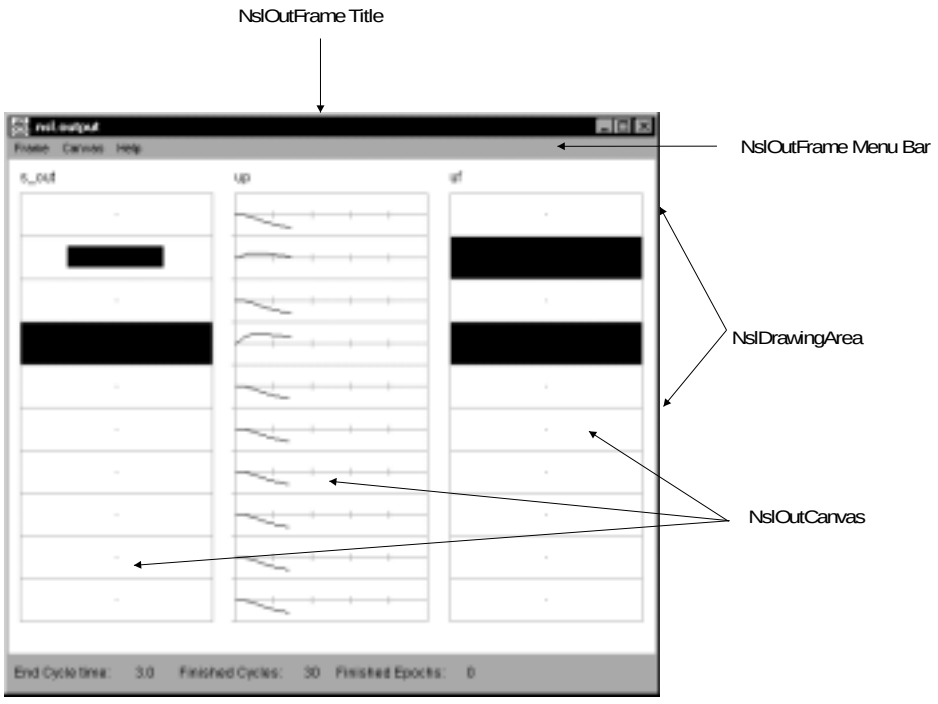

Figure 2.5
MaxSelectorModel NslOutFrame.

The top part of the window contains the title or frame name and the very bottom of the frame contains the **Status line**. The status line displays the current simulation time, finished cycles, finished epochs, and phase. In the middle, the frame contains the **NslDrawingArea**. In this example, the drawing area contains three **NslOutCanvases**: the

first and third corresponds to **Area** graphs while the second corresponds to a **Temporal** graph. (We will describe these graphs in more detail in chapter 5, The User Interface and Graphical Windows.)

2.3 Simulating a Model

If a model is a discrete-event or discrete-time model, the model equations explicitly describe how to go from the state and input of the network at time t to the state and output after the event following t is completed, or at time t+1 on the discrete time scale, respectively. However, if the model is continuous-time, described by differential equations, then simulation of the model requires that we replace the differential equation by some discrete-time, numerical method (e.g., Euler or Runge-Kutta) and choose a simulation time step Δt so that the computer can go from state and input at time t to an *approximation* of the state and output at time $t+\Delta t$. In each case, the simulation of the system proceeds in steps, where each *simulation cycle* updates every module within the model once.

In simulating a model, a basic simulation time step must be chosen. Simulation involves the following aspects of model interaction: (1) simulation control, (2) visualization, (3) input assignment and (4) parameter assignment.

Simulation Control Simulation control involves the execution of a model. The Executive window's "Simulation," "Train" and "Run" menus contain options for starting, stopping, continuing and ending a simulation during its training and running phase, respectively.

Visualization Model behavior is visualized via a number of graphics displays. These displays are drawn on canvases, **NslOutCanvas**, each belonging to a **NslOutFrame** output frame. Each **NslOutFrame** represents an independent window on the screen containing any number of **NslOutCanvas** for interactively plotting neural behavior or variables in general. NSL canvases can display many different graph types that display NSL numeric objects—objects containing numeric arrays of varying dimensions. For example the **Area** graph shown in figure 2.5 displays the activity of a one-dimensional object at every simulation cycle. the size of the dark rectangle represents a corresponding activity level. On the other hand, the **Temporal** graph shown displays the activity of a one-dimensional objects as a function of time (in other words, it keeps a history).

Input Assignment Input to a model varies both in terms of the particular model but also in terms of how it is specified. NSL supports input as script commands in the NSLS language using the **Script Window**, by loading script files, as well as by custom-designed input windows.

Parameter Assignment Simulation and model parameters can be interactively assigned by the user. Simulation parameters can be modified via the "Options" menu while model parameters are modified via the **Script Window**. Additionally, some models may have their own custom-designed window interfaces for parameter modification.

The remaining sections of this chapter illustrate model simulation starting with the *Maximum Selector* model then with *Hopfield* and finally with *Backpropagation*.

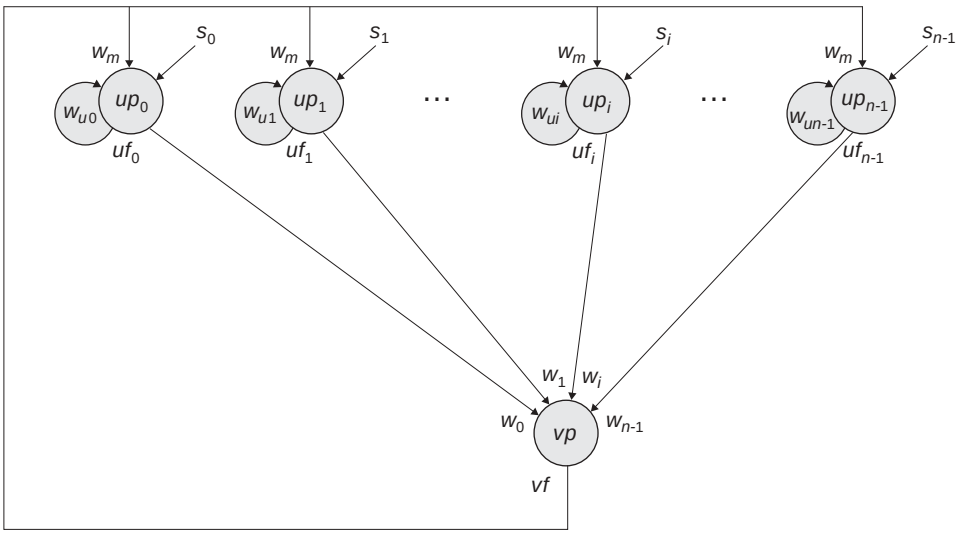

Figure 2.6
The neural network architecture for the Maximum Selector (Didday 1976; Amari and Arbib 1977) where s_i represents input to the network, up_i and vp represent membrane potentials while uf_i and vf represent firing rates. w_m, w_{ui}, and w_i correspond to connection weights.

2.4 Maximum Selector

The *Maximum Selector* neural model (Amari and Arbib 1977) is an example of a biologically inspired neural network. The network is based on the *Didday* model for prey selection (Didday 1976) and is more generally known as a *Winner Take All* (*WTA*) neural network. The model uses competition mechanisms to obtain, in many cases, a single winner in the network where the input signal with the greatest strength is propagated along to the output of the network.

Model Description

The *Maximum Selector* neural network is shown in figure 2.6. External input to the network is represented by s_i (for $0 \le i \le n$-1). The input is fed into neuron u, with up_i representing the membrane potential of neuron u while uf_i represents its output. uf_i is fed into neuron v as well as back into its own neuron. vp represents the membrane potential of neuron v which plays the role of inhibitor in the network. w_m, w_{ui}, and w_i represent connection weights, whose values are not necessarily equal.

The neural network is described by the following set of equations,

$$\tau_u \frac{du_i(t)}{dt} = -u_i + w_u f(u_i) - w_m g(v) - h_1 + s_i$$
$$\tau_v \frac{dv}{dt} = -v + w_n \sum_{i=1}^{n} f(u_i) - h_2 \tag{2.1}$$

where w_u is the self-connection weight for each u_i, w_m is the weight for each u_i for feedback from v, and each input s_i acts with unit weight. wn is the weight for input from each u_i to v. The threshold functions involve a *step* for $f(u_i)$

$$f(u_i) = \begin{cases} 1 & u_i > 0 \\ 0 & u_i \le 0 \end{cases} \tag{2.2}$$

and a *ramp* for $g(v)$

$$g(v) = \begin{cases} v & v > 0 \\ 0 & v \le 0 \end{cases} \tag{2.3}$$

Again, the range of i is $0 \le i \le n$-1 where n corresponds to the number of neurons in the neural array u.

Note that the actual simulation will use some numerical method to transform each differential equation of the form $\tau\, dm/dt = f(m,s,t)$ into some approximating difference equation $m(t+\Delta t) = f(m(t), s(t), t)$ which transforms state $m(t)$ and input $s(t)$ at time t into the state $m(t+\Delta t)$ of the neuron one "simulation time step" later.

As the model equations get repeatedly executed, with the right parameter values, u_i values receive positive input from both their corresponding external input and local feedback. At the same time negative feedback is received from v. Since the strength of the negative feedback corresponds to the summation of all neuron output, as execution proceeds only the strongest activity will be preserved, resulting in many cases in a "single winner" in the network.

Simulation Interaction

To execute the simulation, having chosen a differential equation solver (approximation method) and a simulation time step (or having accepted the default values), the user would simply select "Run" from the NSL Executive's Run menu as shown in figure 2.7. We abbreviate this as **Run→Run**.

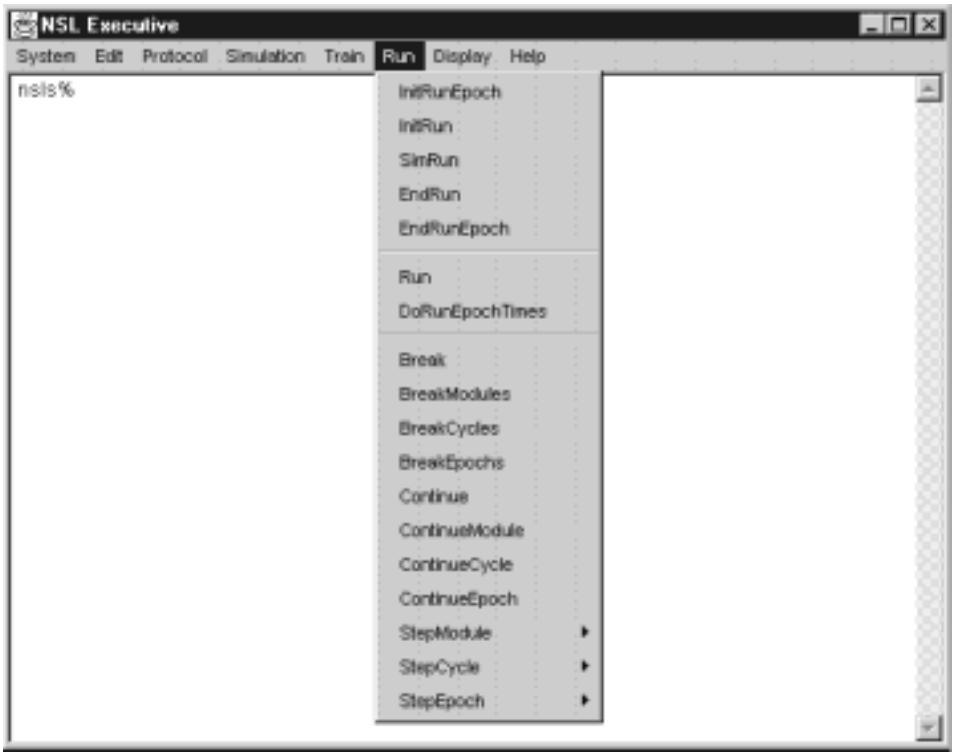

Figure 2.7
The "Run → Run" menu command.

The output of the simulation would be that as shown in figure 2.8.

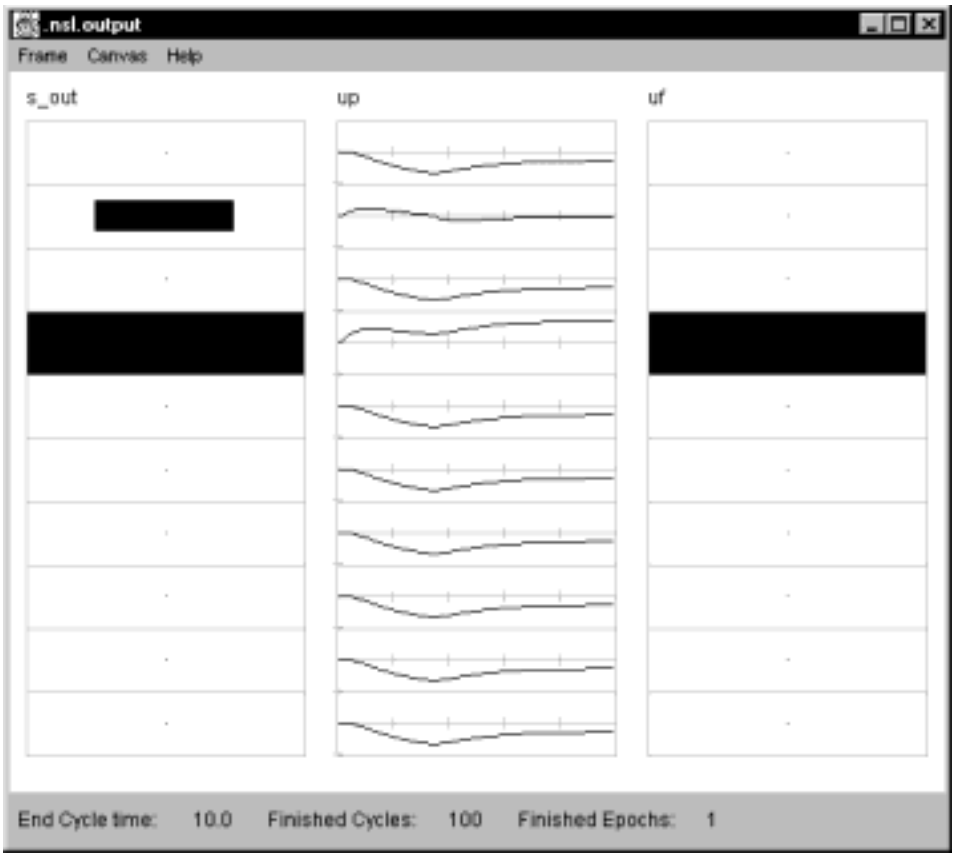

Figure 2.8
Output of the MaxSelectorModel. Notice that the second and fourth elements in the *up* membrane potential layer are affected by the input stimuli; however, the "winner take all" circuit causes the fourth element to dominate the output, as seen in the firing rate, *uf*.

The resulting written output is displayed in the Executive window's shell, as shown in figure 2.9.

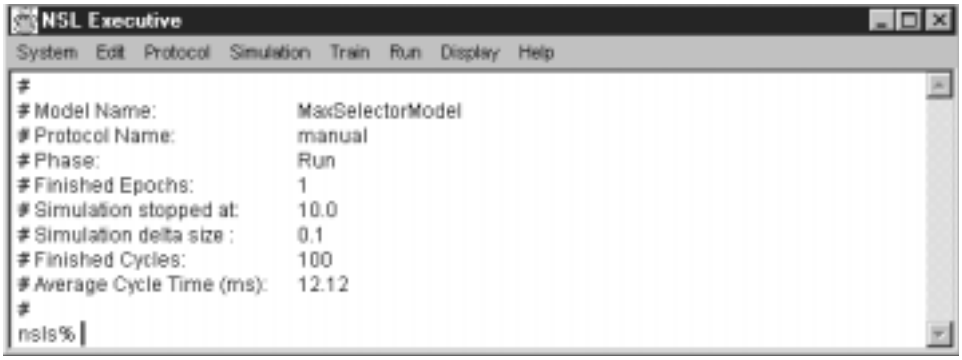

Figure 2.9
Executive window showing the status from Maximum Selector execution.

Recall that NSLS is the NSL scripting language in which one may write a script file specifying, e.g., how to run the model and graph the results. The user may thus choose to create a new script, or retrieve an existing one. In the present example, the user gets the system to load the NSLS script file containing preset graphics, parameters, input and simulation time steps by selecting "**System→Nsls file …**," as shown in figure 2.10.

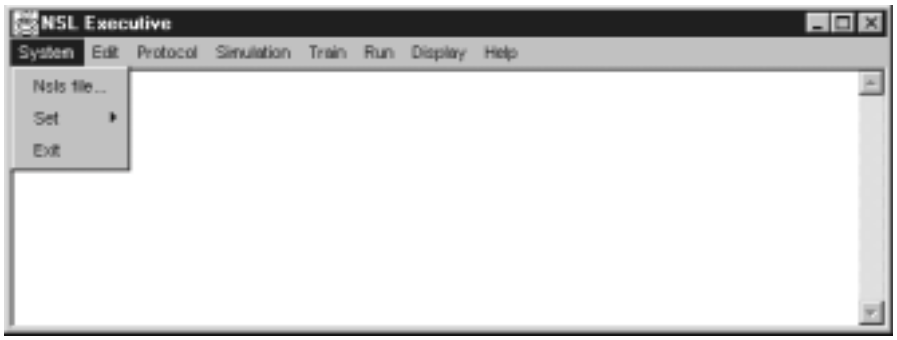

Figure 2.10
Loading a "NSLS" script file into the Executive.

From the file selection pop-up window we first choose the "nsl" directory and then **MaxSelectorModel**, as shown in figure 2.11. Alternatively, the commands found in the file could have been written directly into the **Script Window** but it is more convenient the previous way.

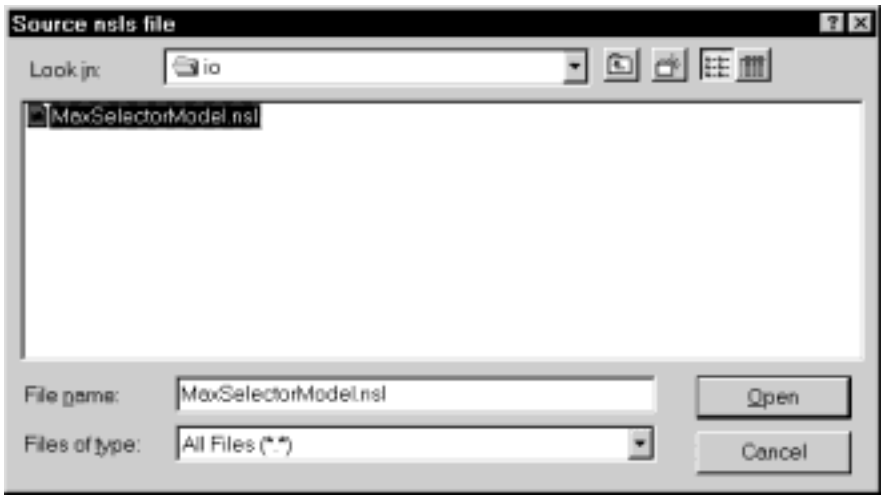

Figure 2.11
The MaxSelectorModel script loaded into the Executive.

Simulation Control

Simulation control involves setting the duration of the model execution cycle (also known as the delta-t or simulation time step). In all of the models we will present, we will provide default values for the simulation control parameters within the model. However, to override these settings the user can select from **System→Set→RunEndTime** and **System→ Set→RunDelta** as shown in figure 2.12.

Figure 2.12
Setting system control parameters.

A pop-up window appears showing the current parameter value that may be modified by the user. In this model we have set the *runEndTime* to 10.0, as shown in figure 2.13, and *runDelta* to 0.1 giving a total of 100 execution iterations. These values are long enough for the model to stabilize on a solution.

Figure 2.13
RunEndTime parameter setting.

To execute the actual simulation we select "Run" from the "Run" menu, as we did in figure 2.7.

The user may stop the simulation at any time by selecting the "Run" menu and then selecting "Break." We abbreviate this as **Run→Break**. To resume the simulation from the interrupt point select **Run→Continue.**

Visualization

The model output at the end of the simulation is shown in figure 2.8. The display shows input array *sout* with an **Area** type graph, i.e., the area of the black rectangle codes the size of the corresponding input, while array *up*, with a **Temporal** type graph, shows the time course for *up*. The last canvas shows another **Area** type graph for *uf* at the end of the simulation. The largest input in *sout* determines the only element of *sout* whose activity is still positive at the end of the simulation as seen in *uf*—the network indeed acts as a maximum selector.

Input Assignment

The *Maximum Selector* model example is quite simple in that the input *sout* is constant. In the example chosen, is consists of only two different positive values (set to 0.5 and 1.0) while the rest are set to zero (total of 10 elements in the vector). In general, input varies with time. Since input is constant in the present case, it may be set similarly to any model parameter. To assign values to parameters, we use the "nsl set" command followed by the variable and value involved. For example, to specify all ten-element values for *sout* we would do:[1]

```
nsl set maxSelectorModel.stimulus.sout { 0 0 0 1 0 1 0 0 0 0 }
```

Since all variables are stored within modules, being themselves possibly stored in other modules until reaching the top level model, it is necessary to provide a full "path" in order to assign them with new values. (These hierarchies will be made clear in chapter 3. For the moment simply provide the full specified path.) Note that arrays are set by specifying all values within curly brackets. Individual array elements may be set by using parentheses around a specific array index, e.g. to set the value of only array element 3 we would do (array indices starting with 0):

```
nsl set maxSelectorModel.stimulus.sout(3) 1
```

As previously mentioned, this model is atypical in that the input is constant. In general, input varies with time as will be shown in most of the other models in the book. If we are dealing with dynamic input we have different alternatives for setting input. One is to specify a "nsl set" command with appropriate input values every time input changes. Another alternative is to specify the input directly inside the model description or through a custom interface. Both *Hopfield* and *Backpropagation* models give examples on how to

dynamically modify input at the script level and through the use of training files described as part of the model definition, respectively. On the other hand, *the Adaptation* model and the *Crowley* model appearing in the second section of the book are examples that set up their input and parameters through custom-designed windows.

Parameter Assignment

Parameters whose values were not originally assigned in the model description, or that we may want to modify, are specified interactively. Two parameters of special interest in the model are the two thresholds, *hu* and *hv*. These two parameters are restricted as follows, $0 \leq hu$, and $0 \leq hv < 1$. (For the theory behind these restrictions, see Arbib, 1989, Sec.4.4.) Their initial values are set to 0.1 and 0.5 respectively. These parameters have their values specified with the "set" command followed by the variable and value involved

```
nsl set maxSelectorModel.maxselector.u1.hu 0.1
nsl set maxSelectorModel.maxselector.v1.hv 0.5
```

To exercise this model the reader may want to change both the input and parameter values to see different responses from the model. We suggest trying different combinations of input values, such as changing input values as well as specifying different number of array elements receiving these values. In terms of parameters we suggest changing values for *hu* and *hv*, including setting them beyond the mentioned restrictions. Every time parameters or input changes, the model should be reinitialized and executed by selecting the "run" menu option.

2.5 Hopfield

Hopfield networks (Hopfield 1982) are recurrent networks in that their complete output at one time step serves as input during the next processing cycle. These networks rely on locally stable states or *attractors* enabling the association of a particular input pattern to one of its "remembered" patterns. These networks are also known as *associative memories* since they will in many cases transform the input pattern into one of the stored patterns (encoded in the network weights) that it best approximates. Unlike the *Maximum Selector*, a *Hopfield* network involves two processing phases—the training phase where synaptic weights are set to desired values and the running phase where the initial state of each neuron is set to the input pattern being tested.

Hopfield networks have been applied to problems such as optimization as in the famous "Traveling Salesman Problem" (Hopfield and Tank, 1985) where given a number of cities a salesman must choose his travel route in order to minimize distance traveled. In general, it may be quite challenging to go from the specification of an optimization problem to the setting of weight matrices to control memory states of a neural network which will "solve" the problem. This becomes more difficult as the number of inputs, cities in this case, increases. (Due to this difficulty, the "Traveling Salesman Problem" has sometimes been called the "Wandering Salesman Problem"!) What makes the matter worse is that this "solution" may only be *locally* optimal, i.e., it may be better than any similar solution yet not as good as some radically different solution. Attempts to find algorithms that produce better than local optimal solutions (e.g., the introduction of noise) have attracted much effort in the neural networks literature, but lie outside our present concern—to demonstrate NSL simulation of Hopfield networks. Besides optimization, *Hopfield* networks have been used in other practical applications such as error-correcting codes, reconstruction, and pattern recognition. The example presented in this section will be a *Hopfield* network for recognizing letter patterns.

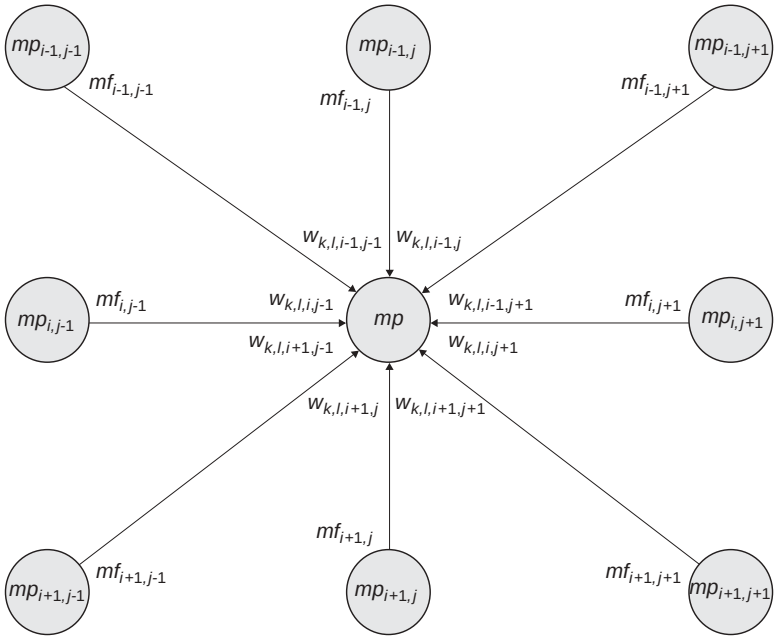

Figure 2.14
The Hopfield network is fully connected with the restriction that no unit may connect to itself.

Model Description

A *Hopfield* network is a discrete-time model consisting of a group of neurons projecting to all other neurons in the network with the restriction that no neuron connects to itself and weights are *symmetric* throughout the network, as shown in figure 2.14. The *Hopfield* model is based on *asynchronous updating* of states: only a single unit, randomly chosen, has its state updated at any given time. As a result the state of the chosen unit may change to reflect prior changes in the states of other units or may remain the same if those changes "cancel out."

The image to be processed does *not*, as might be expected, provide input to the network throughout processing. But rather the input pattern is used to set the initial states of the neurons of the network. To this end, we use double indexing for units m in order to make each unit correspond to a single picture element in a two-dimensional image. The dynamics of the network is then to convert the original pattern into some desired transformation thereof. Each element in the connection matrix w is then specified through four indices. If w_{klij} is the connection between unit m_{ij} and unit m_{kl}, then the activity mp_{kl} of unit m_{kl} is computed directly from the input from all other connections where mf_{ij} is the output from neuron m_{ij}. The computation is given by

$$mp_{kl}(t+1) = \sum_i \sum_j w_{klij}\, mf_{ij}(t) \tag{2.4}$$

Note that unlike the leaky integrator model, the state of a neuron in this discrete-time model does not depend on its previous state—it is completely determined by the input to the neuron at that time step. For our example, we concentrate on binary *Hopfield* networks using discrete neurons whose values can be either +1 or -1. The state of a neuron is given by

$$mf_{kl} = \begin{cases} 1 & \text{if } mp_{kl} \geq 0 \\ -1 & \text{if } mp_{kl} < 0 \end{cases} \tag{2.5}$$

To analyze the network, Hopfield (1984) suggested viewing the network as minimizing an *energy function E* given by

$$E = -\frac{1}{2}\sum_k \sum_l \sum_i \sum_j w_{klij} mp_{kl} mp_{ij} \tag{2.6}$$

Each term is composed of the state of the m_{ij} unit, the state of the m_{kl} unit and the strength of the connection w_{klij} between the two units. (Sophisticated readers will note that each neuron has threshold zero.) This energy function may be interpreted as a measure of constraint satisfaction in the network. If we consider that neurons represent hypotheses in a problem, with an assertion of the hypothesis seen as corresponding to the +1 state of a neuron, and connection weights encode constraints between hypotheses, then the energy function is chosen to be a measure of the overall constraint violation in the current hypotheses. A low energy state would correspond to a state of maximum agreement between pairs of coupled assertions, while energy would increase when states become in disagreement. So long as the weights w_{klij} are symmetric, $w_{klij} = w_{lkji}$, some simple algebra (omitted here) shows that changes in state during asynchronous updates always decrease the energy of the system. Of course, if the "update" of a neuron leaves its state unchanged, then the state of the whole system and thus its energy also remain unchanged. Because all terms are finite there is an energy lower bound in the system and the energy function must have at least one minimum, although many minima may exist. As the system evolves with time, its energy decreases to one of the minimum values and stays there since no further decreases are possible. These points are known as *local energy minima*—we say that they are *attractors* because states move as if attracted to them; once at an energy minimum, the state of the network remains there, so we may also speak of these as fixed points. We can arrange the network in such a way that the desired associations occupy low energy points in state space so that the network will seek out these desired associations. In the present section, we look at a network such that we present noisy images and get back the image that most resembles it by comparing corresponding fixed points.

The key to defining a *Hopfield* network is in choosing the weight matrix. In the present image processing example, we initialize the synaptic weights of the network using a given set of input vectors, i.e., n exemplars pat_m for $0 \le m < n$. We define the weight matrix *w* as

$$w_{klij} = \begin{cases} 0 & k = i, l = j \\ \sum_m pat_{mkl}\, pat_{mij} & \text{otherwise} \end{cases} \tag{2.7}$$

for all *n exemplars* or training patterns in the network. If the input vectors are orthogonal (i.e., their scalar product is 0) then *Hopfield* guarantees that each exemplar becomes a fixed point of the network. (The mathematical justification requires some simple linear algebra. See, e.g., Section 8.2 of Arbib 1989.)

Simulation Interaction

We start the simulation interaction by selecting the *Hopfield* model by selecting "HopfieldModel.nsl" as shown in figure 2.15 (after selecting "**system→Nsls file…**" as shown in figure 2.10).

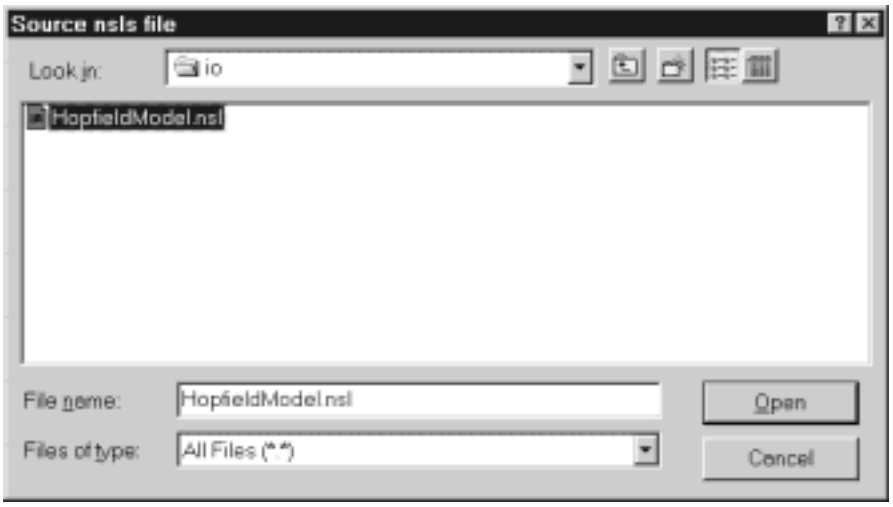

Figure 2.15
The Hopfield model opened by selecting *HopfieldModel.nsl* from the "io" directory.

The example we have chosen is a pattern recognition problem where we train the network to remember letters A, B, C, D and E, as shown in figure 2.16. During testing we shall use one of these letters or a similar pattern as input. We have designed the particular patterns for each letter trying to keep *orthogonality* between them, that is, they are as distinct as possible. This is an important requisite in *Hopfield* networks for good association.

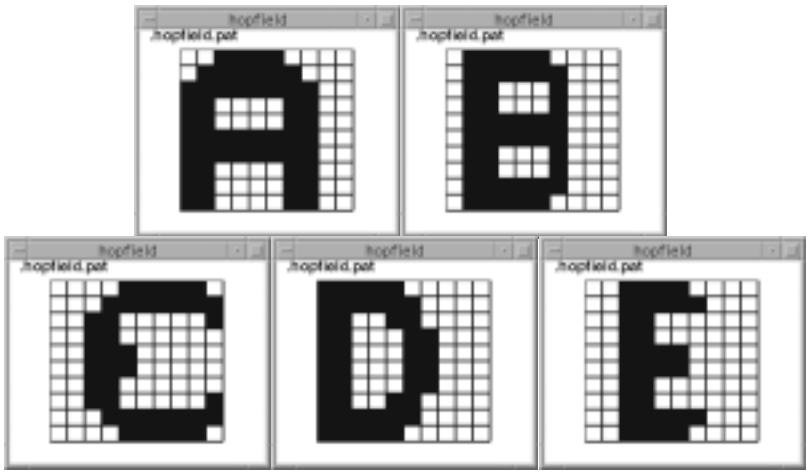

Figure 2.16
Letter A, B, C, D and E, used for setting the connection weights of the Hopfield network. Here we indicate the connections for a typical neuron.

Simulation Control

Two simulation phases, for training and running, are involved in the *Hopfield* model as opposed to the single one in the *Maximum Selector* model. The training phase in *Hopfield* is unusual in that connection weights are not learned but adjusted directly from input patterns, as opposed to the training phase in most other training algorithms such as *Back-propagation*. We set *trainEndTime* to 1.0, and also *trainDelta* to 1.0, giving a total of 1 iteration through all the patterns. Additionally, the train cycle is executed for a single *epoch*, a single pass over all training patterns, thus we set *trainEpochSteps* to 1 as well. All the control commands are set in the "hopfield.nsl" file, including specification of the five letters, A, B, C, D and E chosen for the example.

Once all letters have been read we are ready to execute the run phase indefinitely until a stable solution is reached. Depending on the test letter the solution may take a different number of time steps. Thus, the model will stop running only when the solution has stabilized, in other words, when output for a new time step would yield exactly the

same output as in the previous time step. To achieve this, we set *runEndTime* to 5000 (corresponding to protracted execution; alternatively, we could specify that detection of a suitable period of constant internal states makes it stop) and *runDelta* is set to 1.0 (this value is arbitrary in discrete-time models). To execute the running phase we select "Run" from the Run menu. To start processing all over again we would execute "Simulation→ initModule" followed by the training phase and then the run phase.

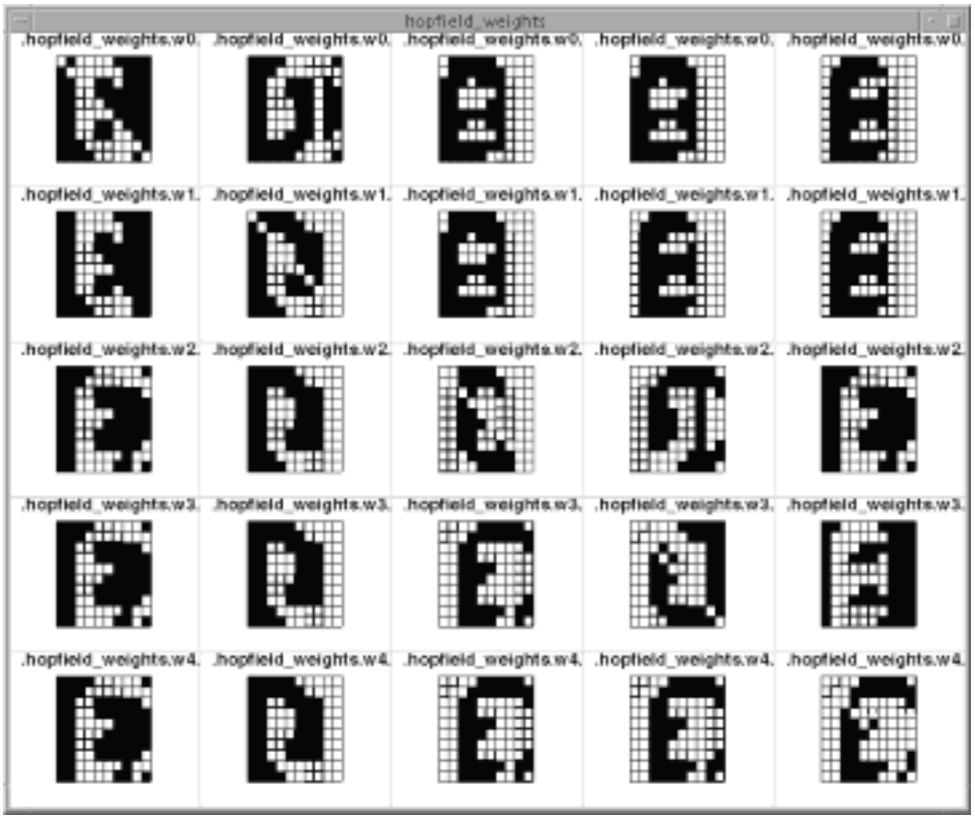

Figure 2.17
The figure presents the 5x5x10x10 weight array after training the Hopfield network. The 5x5 array organization represents the twenty-five 10x10 sub-matrices.

Visualization

The stored script file generates a number of display frames. We show in figure 2.17 the matrix of connection weights that you should obtain after training the model with letters A, B, C, D and E.

Once the network has completed the training cycle we input different letters to recall the memorized letter closer to it. We first try the model by recalling letters from the original ones, as shown in figure 2.18.

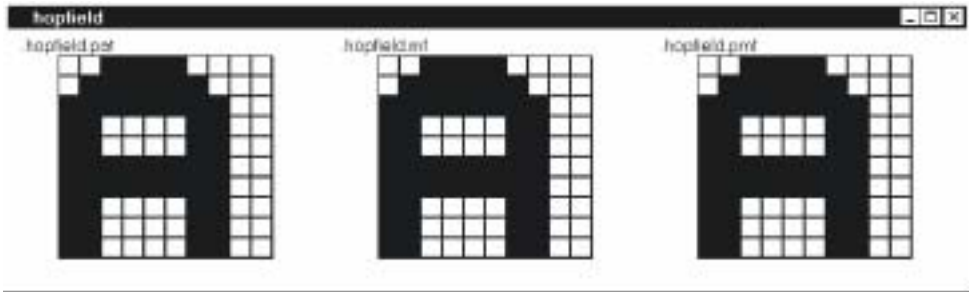

Figure 2.18
The top portion of the figure shows the input letter A, while the lower portion shows the output at the end of the simulation. In this simpler case letter A is recalled exactly as presented.

We show the energy as a function of time in figure 2.19, notice how it goes down as the network settles into a solution.

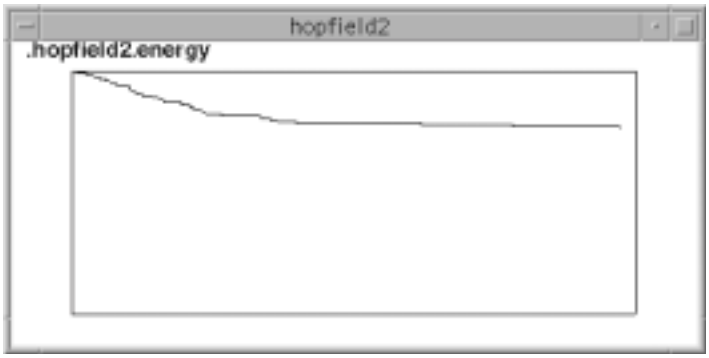

Figure 2.19
Energy as a function of time.

The network is able to recall correct answers from noisy versions of these letters. For example, the input image shown in figure 2.20 would recall letter A. Watch how the isplay reveals the cleaning up of the noisy image.

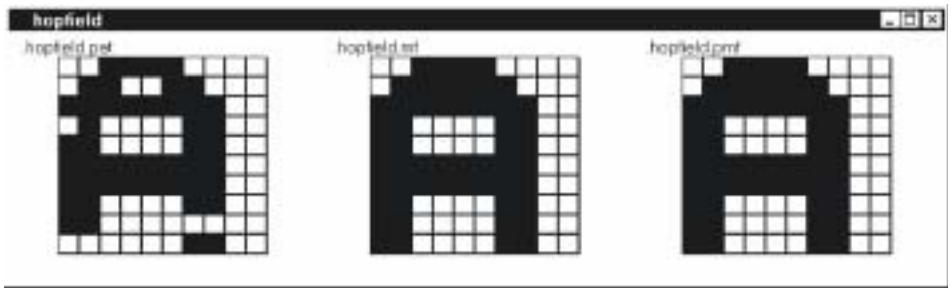

Figure 2.20
Recalling letter A from noisy image.

We can also input a letter such as an F that closely resembles letter E in the training set, as shown in figure 2.21.

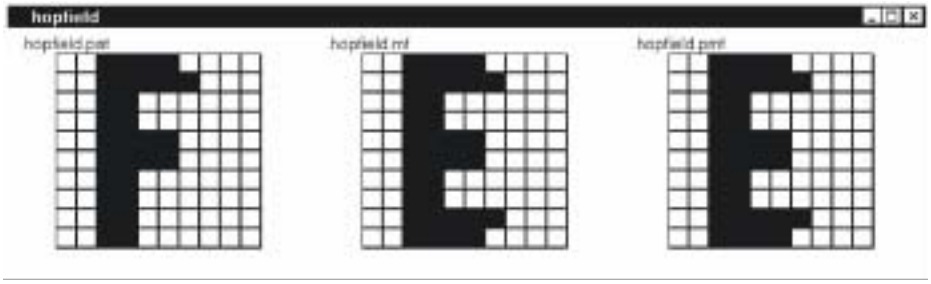

Figure 2.21
Recalling letter E from letter F, the closest to it in the training set.

In some cases the network may "remember" patterns that were not in the original set of examples, as shown in figure 2.22. These are called *spurious* states, unexpected valleys or local minima in the energy function, an unavoidable feature of *Hopfield* networks where processing is "stuck" in intermediate undefined states.

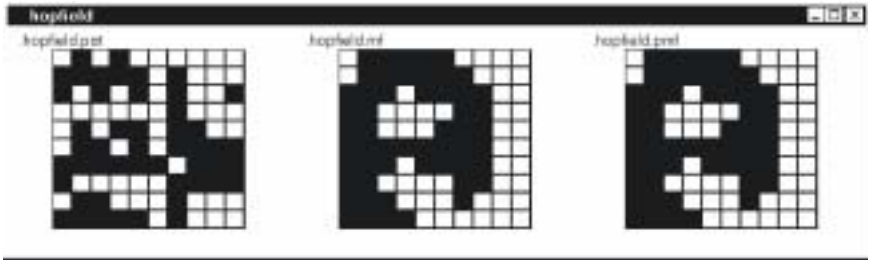

Figure 2.22
Spurious state of the Hopfield network.

This aspect exemplifies one of the shortcomings of *Hopfield* networks in terms of its tendency to stabilize to a local rather that a global minimum of the energy function. Another shortcoming relates to the capacity of *Hopfield* networks is that its capacity goes

down as the number of stored patterns increases beyond some critical limit. This results in *crosstalk* from nonorthogonal patterns causing *attractors* to wander away from the desired locations. As more nonorthogonal patterns are stored, the more likely errors become (Abu-Mustafa and St. Jacques 1989). Hopfield (1982) has shown that if more patterns are stored than 15% of the number of units in the network (in our example 15 patterns, compared to 100 units in total), the system randomizes its energy minima. In other words, above this critical value the retrieved pattern has no relation to the stored pattern. (Of course, if there are 100 neurons, then one can store 100 orthogonal patterns. However, "real" patterns such as the letters of the alphabet are very unlikely to form an orthogonal set of vectors. Thus the mathematical results are based on expected performance when vectors are chosen at random. The point here is that if a few vectors are chosen at random, with each "pixel" as likely to be on as off, their pairwise scalar products will be close to zero, but this becomes more and more unlikely as the number of patterns increases. The surprise, to people unacquainted with critical phenomena in statistical mechanics, is that there is a critical number of patterns at which quasi-orthogonality breaks down, rather than a slow degradation of performance as the number of patterns increases.)

Input Assignment

Input plays an important and delicate role in the model. During training, network weights are set according to input matrices representing letters to be remembered. During an execution or simulation run, the network is given an input matrix to be associated with one of its remembered states that best matches the pattern.

In the *Maximum Selector* model we showed how we set constant input, in a manner similar to parameter assignment. In the training phase of the Hopfield model, we need dynamic input to read in a sequence of n input patterns. In the present model, these do not function as neural network inputs (as might happen if we modeled an explicit learning model) but instead serves as input for a process that computes weights according to equation (2.7). Training the *Hopfield* model thus requires dynamic input. We read in the n training patterns by calling the "nsl set" command multiple times. In the example each letter corresponds to a 10x10 matrix. For example, letter "A" is defined as follows:

```
nsl set HopfieldModel.input.out {
{ -1 -1 1 1 1 1 -1 -1 -1 -1 }
{ -1 1 1 1 1 1 1 -1 -1 -1 }
{ 1 1 1 1 1 1 1 1 -1 -1 }
{ 1 1 -1 -1 -1 -1 1 -1 -1 -1 }
{ 1 1 -1 -1 -1 -1 1 1 -1 -1 }
{ 1 1 1 1 1 1 1 1 -1 -1 }
{ 1 1 1 1 1 1 1 1 -1 -1 }
{ 1 1 -1 -1 -1 -1 1 1 -1 -1 }
{ 1 1 -1 -1 -1 -1 1 1 -1 -1 }
{ 1 1 -1 -1 -1 -1 1 1 -1 -1 }}
```

Note the curly brackets separating matrix rows. The rest of the letters are defined in a similar way. In order to control the input in a dynamic way we set the input from the script window for each letter being computed by the weight assignment equation followed by the Train command (performs initTrain, simTrain repeated, endTrain) with each epoch incrementing the expressions in Equation (2.7) by adding in the terms corresponding to the current pattern $patt_m$.

```
nsl train
```

We now turn to the "input" for the Running phase. As we have seen, a Hopfield network does not have input in the conventional neural network sense. Instead, the "input" sets the initial state of the network, which then runs to equilibrium, or some other halting condition. The "output" for this particular run is taken from the final state of the network. In each run phase of a simulation, we set the "input" to any arbitrary pattern (i.e., it will probably not belong to the training set) and then run the network as many cycles as necessary. We shall look at the details for defining this model in chapter 3, The Modeling Overview.

Parameter Assignment

There are no parameters that need to be adjusted in the model. Being a discrete-time model, *Hopfield* updates the state directly from its current input and state. Unlike the leaky integrator, there are no time constants. Weights are computed by the training phase and neuron thresholds are set to zero.

You may exercise the model by modifying both the test-input patterns as well as the patterns used for training. They do not even have to be letters.

2.6 Backpropagation

Backpropagation (Werbos 1974; Rumelhart et al. 1986) is an especially widespread neural network architecture embodying *supervised learning* based on gradient descent ("hill climbing" in the downward direction). Supervised learning involves a training set representing both the given problem and the corresponding solution defined as a set of (input, target) training pairs. The goal of successful learning is to acquire general knowledge about the data or training set so the network can use it to resolve similar problems it has not seen before. There are two important factors in building a successful *backpropagation* network: the *training set* and the *network configuration*.

The *training set* consists of a number of training pairs where each pair (input, target) contains a target vector that is deemed the correct response to its corresponding input vector. A "supervisor" compares the resulting network output for a given input vector against the target vector to produce an error. This error is then used to drive the adjustment of weights in the network in such a way that the error is reduced to its minimum. The process of error minimization consists of following a steep path down the input-output error function. Although there is no guarantee of minimizing all errors (*gradient descent* may only find a local minimum, like a valley high in the hills, as for *Hopfield*), a *backpropagation* network is usually able after many training cycles to reduce the errors to a satisfying degree.

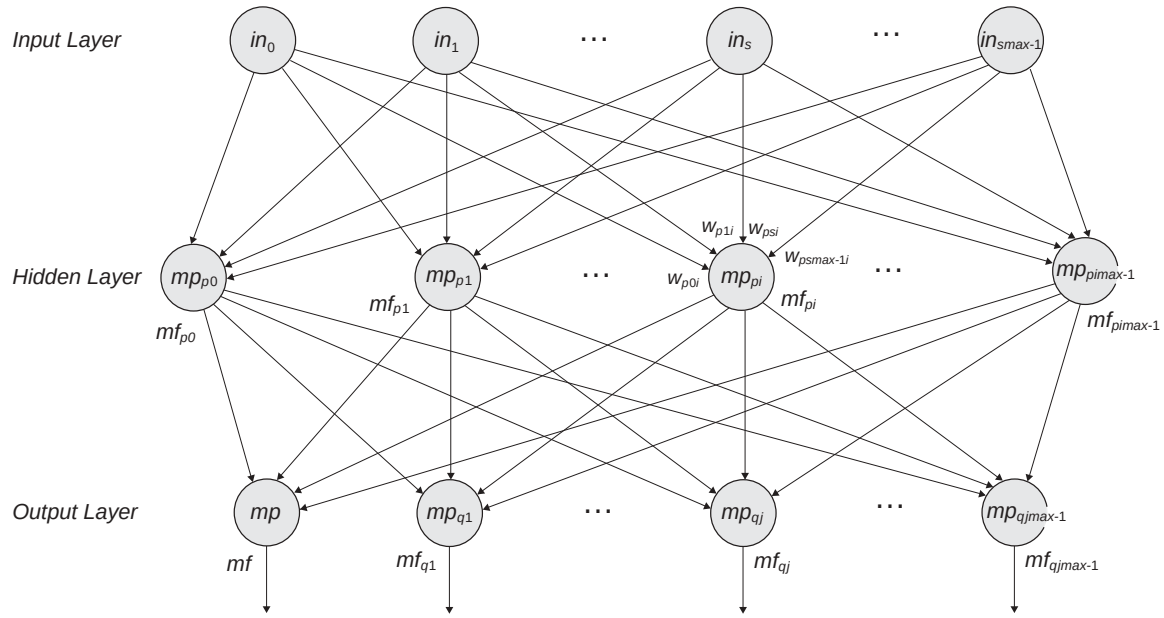

Figure 2.23
The Backpropagation network architecture is made of an input layer connected to a hidden layer that is then connected to an output layer. Units are fully connected between layers without any interconnection to other units in the same layer.

The *network configuration* consists of neurons organized into at least three different layers: an input layer, one or more hidden or middle layers and an output layer (figure 2.23). The network processes information in two distinct modes, a *feedforward* and a *backpropagation* mode. The *feedforward* mode is just the normal mode of operation of a neural network without loops: activity is fed forward from one layer to the next (input to hidden layer, hidden to additional hidden layers if more than one exists, and finally hidden to output layer). There are no loops in strong contrast to the fully recurrent *Hopfield* network. In the *backpropagation* mode, learning propagates backwards by adjusting synaptic weights from output to input layers. The most common configuration is a three-layer network with all possible connections from each layer to the next. Implementing four or more layers is usually discouraged because of the computational burden of the *backpropagation* training process. Both mathematical proof and practical uses of *backpropagation* show that three-layer networks are sufficient for solving most problems (Rumelhart, et al. 1986).

In designing the network configuration, the most important parameter is the network size and the number of units used in the hidden layer to represent features of the problem. There are tradeoffs to consider. With too large a number of hidden units, the network will have the ability to *memorize* each element of the training set separately, and thus will not generalize well. With too small a number of hidden units, there may not be enough memory to store the knowledge (refer to Smith (1993) on how to build appropriate network configurations).

Backpropagation has been applied to a large number of applications in many domain areas, from handwriting recognition and speech synthesis to stock market prediction and on.

Model Description

As we have seen, *Backpropagation* is a typical multi-layer neural network model consisting of an input layer, hidden or middle layer(s), one in this case, and an output layer (figure 2.23). The network is fully connected from one layer to the next, but lacks any connectivity between neurons belonging to the same layer, or back to previous layers.

The *BackPropagation* algorithm works in two phases, as in *Hopfield.* First, a training phase adjusts network weights and then a running phase matches patterns against those

already learned by the network. However, these two phases are not to be confused with a *feedforward* and a *backpropagation* modes introduced above. The training phase is made up of a large number of learning cycles, each comprising a forward pass (*feedforward* mode) and backward pass (*backpropagation* mode). The running phase is made of a single forward pass taking a single cycle although sharing the same forward pass equations (*feedforward* mode) as in the training phase.

Feedforward Mode

During the feedforward mode, the network reads an input vector that is fed into the input layer. The input layer does not do any computation on the input pattern and simply sends it out to the hidden layer. Both the hidden layer and output layer have their neuron activity (corresponding to the membrane potential in more biologically oriented models) defined as a direct summation of all inputs to the neuron multiplied by their respective weights. In the model, *in* represents a unit in the input layer, mp_p represents a neuron in the hidden layer and mp_q a neuron in the output layer.

Hidden Layer

The membrane potential mp_p for a neuron in the hidden layer receives its activation from the input layer multiplied by the respective weights, as described next.

$$mp_p = \sum_s w_{sp} in_s + h_p \tag{2.8}$$

where h_p is the threshold value.

After mp_p is computed, an activation function is used to produce the output mf_p.

$$mf_p = f\left(mp_p + h_p\right) = \frac{1}{1 + e^{-\left(mp_p + h_p\right)}} \tag{2.9}$$

where f is a sigmoid function used to compress the range of mp_p so that mf_p lies between zero and one, and e is the mathematical exponential constant. The sigmoid function is used since *Backpropagation* requires the activation function to be everywhere differentiable. The sigmoid function not only satisfies this requirement but also provides a form of automatic gain control. If mp_p is near one or zero, the slope of the input/output curve is shallow and thus provides a low gain. The sigmoid function also has the advantage that large mp_p values will not dominate small mp_p values in influencing the network in going towards the global minimum.

Output Layer

The membrane potential mp_q for a neuron in the output layer receives input from the hidden layer multiplied by the respective weights.

$$mp_q = \sum_p w_{pq} mf_p \tag{2.10}$$

where h_q is the threshold value.

After mp_q is calculated, an activation function is used to produce the output mf_q.

$$mf_q = f\left(mp_q + h_q\right) = \frac{1}{1 + e^{-\left(mp_q + h_q\right)}} \tag{2.11}$$

and the activation function f is similar to that defined for neurons in the hidden layer.

Backpropagation Mode

While the feedforward mode is used during both the training and running phases of the network, the backpropagation mode is only used during training. For each cycle of train-

ing, the simulator reads a pair of input and target vectors from a training data file. The input vector is fed into the input layer for a feedforward computation, while the target vector is set aside for later error computation. After completion of the forward activation flow, each output neuron will receive an error value—the difference between its actual and desired input—from the training manager module, (the training manager module will be discussed in detail in chapter 3, The Modeling Overview).The backpropagation mode then adjusts the weights according to a modified gradient descent algorithm wherein weight adjustment propagates back from the output layer through the hidden layers of the network.

Output Layer

The error is first calculated for the output layer:

$$error_q = desiredOutput_q - actualOutput_q \quad (2.12$$

where *desiredOutput* is obtained from the training file and *actualOutput* is computed by the forward pass output layer firing mf_q.

The accumulated error *tss* is given by the sum of the square of the errors for all neurons of the output layer

$$tss = \sum_t error_q^{\,2} \quad (2.13)$$

To compensate for this error we define δ_q representing the change to be applied to weights and threshold in the output layer given by

$$\delta_q = f'(mp_q) \times error_q \quad (2.14)$$

where $f'(mp_q)$ is the derivative of squashing function *f*. With the simple sigmoid function used, the derivative is:

$$f'(mp_q) = mf_q \times (1 - mf_q) \quad (2.15)$$

The resulting δ_q is then used to modify the thresholds and weights in the output layer as follows

$$\Delta h_q = \eta \delta_q \quad (2.16)$$

$$h_q(t+1) = h_q(t) + \Delta h_q \quad (2.17)$$

$$\Delta w_{pq} = \eta \delta_q \times mf_p \quad (2.18)$$

$$w_{pq}(t+1) = w_{pq}(t) + \Delta w_{pq} \quad (2.19)$$

where

η represents the learning rate parameter corresponding to how fast learning should be.

$h_q(t)$ represents the threshold value for neuron q in the output layer at step t before adjustment is made.

$h_q(t+1)$ represents the threshold value for neuron q in the output layer at step $t+1$ after adjustment.

$w_{pq}(t)$ represents the weight value from neuron p in the hidden layer to neuron q in the output layer at step t before adjustment is made.

$w_{pq}(t+1)$ represents the value of the weight at step $t+1$ after adjustment.

Hidden Layer

Once the errors are computed and threshold and weight updates have taken place in the output layer, the hidden layer errors need to be computed as well. Since there is no explicit teacher for the hidden units, *Backpropagation* provides a solution by propagating

the output error back through the network. To compensate for this error we define δ_p representing the change to be applied to weights and threshold in the hidden layer,

$$\delta_p = f'(mp_p) \times \sum_q \delta_q w_{pq} \tag{2.20}$$

where $f'(mp_p)$ is the derivative of the sigmoid function in neuron p similar to the $f'(mp_q)$ function in the output layer and w_{pq} is the value of the weight from neuron p in the hidden layer to neuron q in the output layer. As before,

$$f'(mp_p) = mf_p \times (1 - mf_p) \tag{2.21}$$

There is a reason why the hidden layer needs to receive the summation of the products of error δ_q multiplied by weight w_{pq}. Since each neuron contributes differently to the output, its share of the error is also different. By associating the error and the weight, each neuron in the hidden layer will be evaluated by its corresponding contribution to the error and corrected accordingly.

The threshold and weight modification equations are similar in computation to that of the output layer, with delta change δ_p used to modify the thresholds and weights in the output layer,

$$\Delta h_p = \eta \delta_p \tag{2.22}$$

$$h_p(t+1) = h_p(t) + \Delta h_p \tag{2.23}$$

$$\Delta w_{sp} = \eta \delta_p \times in_s \tag{2.24}$$

$$w_{sp}(t+1) = w_{sp}(t) + \Delta w_{sp} \tag{2.25}$$

where

η represents the learning rate parameter corresponding to how fast learning should be.

$h_p(t)$ represents the threshold value for neuron p in the hidden layer at step t before adjustment is made.

$h_p(t+1)$ represents the threshold value for neuron p in the hidden layer at step $t+1$ after adjustment.

$w_{sp}(t)$ represents the weight value from unit s in the input layer to neuron p in the hidden layer at step t before adjustment is made.

$w_{sp}(t+1)$ represents the weight value at step $t+1$ after adjustment.

Simulation Interaction

To illustrate an actual example we will train the network to learn an exclusive or (XOR) function, as shown in the table below. This is a simple although illustrative example in that a simpler Perceptron without the hidden layer would not be able to learn this function. The function is shown in table 2.1.

Input	Output
0 0	0
0 1	1
1 0	1
1 1	0

Table 2.1
Training file format.

We turn now to the NSLS commands stored in "BackPropModel.nsl", where the file is loaded into the NSL Executive in order to simulate the model. We load

"BackPropModel.nsl" by selecting the BackPropModel.nsl model as shown in figure 2.24 (after selecting "**system→Nsls file…**" as shown in figure 2.10).

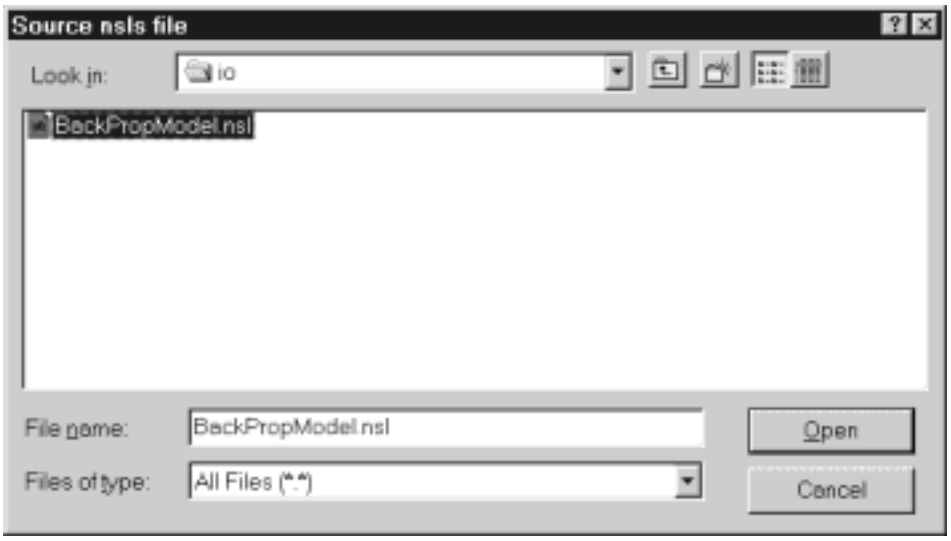

Figure 2.24
Opening the BackPropModel script file.

Simulation Control

As for *Hopfield*, *Backpropagation* requires both the training and running phases. Simulation control for this model involves setting up the duration for both phases as well. The training phase involves multiple cycles. From the script window we set *trainEndTime* to the number of training patterns specified by *numPats* and *trainDelta* to 1.0 in order to have as many training steps as there are training patterns. (These are also the defaults specified in the model code which we will be discussing in chapter 3, The Modeling Overview.) We then set *runEndTime* to 1.0 and *runDelta* to 1.0.

Additionally, the training cycle will be executed for an unspecified number of *epochs*, where every epoch corresponds to a single pass over all patterns. We set *trainEpochSteps* to 5000 telling the system to train almost indefinitely until some suitable ending makes it stop, in this case, when the error (stopError) is small enough. To make the system learn, we issue the **nsl train** command from the script window. As learning keeps progressing, if the total sum of the square error (*tss)* is not satisfactory, the learning rate η can be adjusted. When the *tss* value reaches a very small *stopError*, the network has been successfully trained. At that point we issue the **"nsl source backproprun"** command from the script window. To reinitialize the system after a complete run, we would issue the "nsl initModule" command.

Visualization

The network training error *tss* can be visualized as the network gets trained, as shown in figure 2.25. As the error gets smaller *tss* approaches 0 meaning the network has learned.

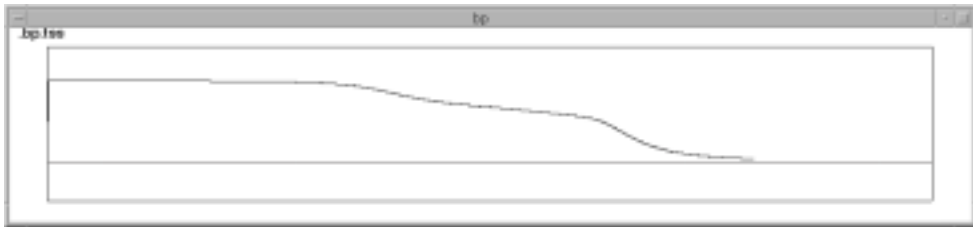

Figure 2.25
The error *tss* is visualized as a temporal graph as the network is training with the XOR example.

Figure 2.26 shows the result of running the trained network with one of the XOR inputs.

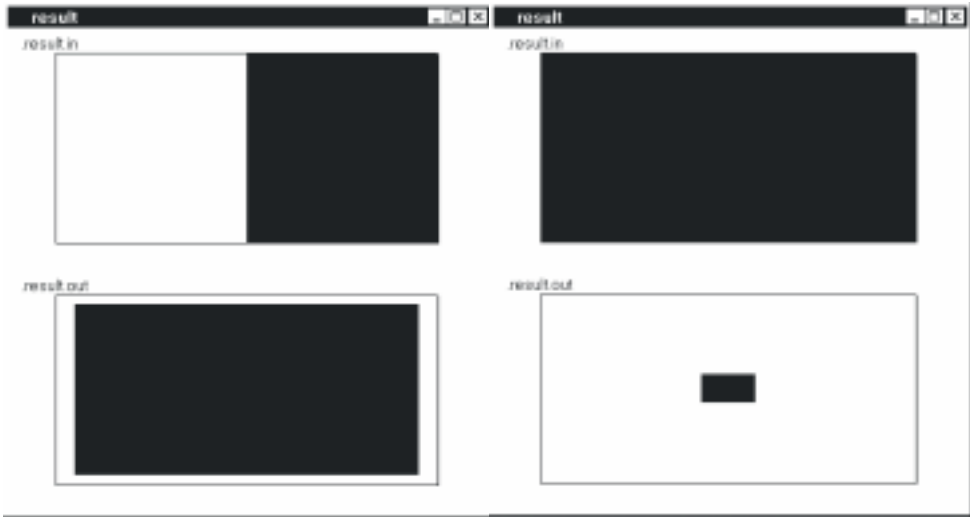

Figure 2.26
The display on the left-hand side shows an input to the network set to "0 1". After the network has been run the output becomes 1, as expected. The display on the right-hand side shows an input to the network set to "1 1" The output this time becomes 0.

Input Assignment

To simplify the training process and to avoid deeper knowledge of NSL, we assign the training set directly to the model as a training array rather than from an external file as is usually the case. (We will show this more "realistic" approach in the NSLM chapter where we go over more extensive details of the modeling language NSLM. Obviously the approach taken will be more involved when dealing with large data sets.) The training set format is shown in table 2.2.

File Format	Example (XOR)
<num_patterns>	4
<input1> <input 2> <output>	0 0 0
<input1> <input 2> <output>	0 1 1
<input1> <input 2> <output>	1 0 1
<input1> <input 2> <output>	1 1 0

Table 2.1
Training file format.

The first row in the file specifies the number of patterns in the file. Training pairs are specified one per row consisting in the XOR example of two inputs `<input1>` and `<input2>` and a single output `<output>`. The training set input is assigned as follows

```
nsl set backPropModel.train.pInput {
{ 0 0 } { 0 1 } { 1 0 } { 1 1 } }
nsl set backPropModel.train.pOutput {
{ 0 } { 1 } { 1 } { 0 } }
```

Note again the curly brackets separating elements in two-dimensional arrays, similar to input in the Hopfield model.

Parameter Assignment

The *Backpropagation* layer sizes are specified within the present implementation of model, i.e., if the number of units in any layer changes, the model has to be modified accordingly and recompiled. The alternative to this example could be to treat layer sizes

as model parameters to be set interactively during simulation initialization. While the latter approach is more flexible since layer sizes tend to change a lot between problems, we use the former one to avoid further complexity at this stage. Thus, the user will need to modify and recompile the model when changing layer sizes. In our example we use 2 units for the input layer, 2 for the hidden layer and 1 for the output layer.

Additionally, we set *stopError* to a number that will be small enough for the network to obtain acceptable solutions. For this example, we use 0.1 or 10% of the output value,

```
nsl set backPropModel.layers.be.stopError 0.1
```

The learning parameter η is represented by the *learningRate* parameter determining how big a step the network can take in correcting errors. The learning rate for this problem was set to 0.8 for both the hidden and output layers.

```
nsl set backPropModel.layers.bh.lRate 0.8
nsl set backPropModel.layers.bo.lRate 0.8
```

The training step or delta is typically set between 0.01 to 1.0. The tradeoff is that if the training step is too large—close to 1—the network tends to oscillate and will likely jump over the minimum. On the other hand, if the training step is too small—close to 0—it will require many cycles to complete the training, although it should eventually learn.

This is obviously a very simple model but quite illustrative of *Backpropagation*. As an exercise we encourage you to try different *learningRates (lRate)* and *stopError* values. Additionally, you can modify the training set although keeping the same structure. In section 3.5 you may try changing the layer sizes in designing new problems. Also, if you are not satisfied with the training, there are two ways to keep it going. One is to issue an **initModule** command, adjust *trainEndTime* to a new value, and then train and run again. The other is to save the weights, issue an **initModule**, load the weights again, and then type *simTrain* at the prompt.

2.7 Summary

In this chapter we have given an introduction to NSL simulation as well as an overview of three basic neural models, *Maximum Selector*, *Hopfield* and *Backpropagation* in NSL. These models, although different, take advantage of a consistent simulation interface provided by NSL.

Notes

1. Currently, we are completing the Numerical Editor Input interface/widget which will allow us to set any writable variable within the model from a pop-up window. The widget will eliminate extra typing in the script window.