SRC TR 88 - 43

BACK PROP: A TOOL for LEARNING ABOUT CONNECTIONIST ARCHITECTURES

by

J. Pollack, M. Evett and J. Hendler

BackProp: A Tool for Learning about Connectionist Architectures

Jordan Pollack
Computer Research Lab
New Mexico State University

Matthew Evett and James Hendler
Systems Research Center
University of Maryland

March 26, 1988

Abstract

This paper provides an implementation, in Common Lisp, of an "epoch learning algorithm," a simple modification of the standard back-propagation algorithm. This implementation is not intended to be a general purpose, high powered back-propagation learning system. Rather, this paper seeks only to provide a simple implementation of a popular and easily understood connectionist learning algorithm. It is intended to be a teaching tool for AI researchers wishing to familiarize themselves or their students with back-propagation in a language with which they are comfortable.

There is currently a great amount of interest in the AI community about connectionist learning algorithms. Back-propagation is among the most readily understood and easily studied of these algorithms. For this reason, researchers interested in exploring connectionist algorithms for the first time might well choose the implementation of a back-propagation algorithm as a starting point. In fact, this algorithm has been implemented at dozens of different labs and colleges across the country. Most of the published implementations, however, are in some language other than Lisp, usually C. Back-propagation is a computationally intensive algorithm, involving lots of number crunching, and that is not usually viewed as Lisp's forte. For those wishing merely to tinker with back-propagation, this is a problem. Most of the interest in these techniques is in the AI community, where the language of choice is Lisp, or in engineering groups, where programming expertise may be lacking.

This paper provides an implementation, in Common Lisp, of an "epoch learning algorithm," a simple modification of the standard back-propagation algorithm. We emphasize that the implementation is not intended to be a general purpose, high powered back-propagation learning system. Rather, this paper seeks only to provide a simple implementation of a popular and easily understood connectionist learning algorithm, and is intended to be a tool for researchers wishing to familiarize themselves with back-propagation in a language with which they are comfortable.

The code is carefully documented and easily modified. Users may run the code as is -- there are functions for creating and running several simple network topologies ("2-2-1 XOR", "4-2-4 Identity", etc.). Utilities are provided for examining the contents and performance of the network. Alternatively, there are several functions for creating new topologies.

The implementation in this paper is an adaptation of Jordan Pollack's InterLisp version. The code was ported, cleaned up and annotated by Evett at the University of Maryland. In addition, several new statistical utilities were added.

Description of Back-Propagation:

The back-propagation algorithm used in this code is a simple modification of that described in Chapter 8 of [Rumelhart & McClelland, 1986] (hereafter referred to as "PDP"), using the "generalized delta rule" as its learning procedure. This section makes several references to the terms and variable names of the PDP description of the algorithm. These references are meant to serve as reference points for readers familiar with PDP's treatment, and such familiarity would be helpful for understanding this system. Readers seeking a fuller understanding of the back-propagation algorithm should see PDP. What follows is only a brief outline of the algorithm.

Back-propagation refers to a class of learning algorithms used to train associative networks to yield certain output patterns when corresponding input patterns are applied to

the net. It is a supervised, iterative, gradient-descent, learning technique. With each presentation of the input patterns, the algorithm alters the weights of the links of the net in such a way as to (hopefully) cause the net to react more correctly (i.e., make the output more similar to the desired output patterns) to the inputs.

Our implementation works only on strictly feed-forward nets. The activation function of the nodes of the nets is sigmoid [PDP, equation 15], mimicking a threshold function with cross-over at 0.0.

The algorithm works as follows: an input pattern is asserted as the output signal of the input nodes of the net. This signal propagates forward through the net until the activation levels of the output nodes stabilize. The activation level of each node is proportional to the sum of the signals received from the nodes inputting to it, and to the weights of the links connecting the node to its inputs.

The actual output pattern is now compared to the desired one. If the difference between the two is "acceptable" (as defined by the user), then no learning occurs. Otherwise, the back-propagation algorithm is used. For example, say the input pattern is [0 0 1] and the desired output pattern is [1 0] (that is, there are three input nodes in the net, and when their activation levels are locked at 0.0, 0.0, and 1.0 respectively, we desire that the two output nodes' activation levels be 1.0 and 0.0.) If the actual output pattern is [0.9 0.15], and "acceptable" is defined as no output value differing by more than 0.2 from its desired value, then the net has performed adequately, and no learning occurs. If the acceptable difference had been only 0.10, though, then learning would occur.

Learning is effected by lessening the weights of incoming links if a node's activation level was too high, or increasing the weights if the level was too low. The difference between the desired and actual activation levels is called the "error signal" (d_{pj}, in PDP). Back-propagation is used to propagate the error signals from the output nodes to the hidden layers of the net so as to determine the error signals of those nodes. This propagation is necessary because the user does not specify the desired activation levels of the intermediate nodes, but only of the output nodes. Back-propagation numerically "assigns blame" to hidden units. The error signal of an internal node is proportional to its own activation level, the error signals of the nodes it outputs to, and the weight of the links to those nodes [PDP, pp. 329-330.]

Because of the form of the equations that calculate the error signals of internal nodes, the back-propagation process is iterative: the error signals are calculated for the output nodes, then for the nodes inputting to the output nodes, then for the nodes inputting to those, etc. Eventually the propagation reaches the input nodes and stops.

After the error signals have been calculated for all the nodes of the net, the algorithm determines by how much to change the weights of the net's links. For a link, this amount (Dw_{ji}, in PDP--we'll call it "delta-w" from now on) is proportional to a constant *learning rate* (h, in PDP), the error signal of its output node, the activation level of its input node, and a *momentum term*--a combination of a *momentum constant* (a, in PDP) and the

amount of change to the link's weight in previous presentations of the input patterns. See [PDP, eqn. 15].

In our implementation the link weights are not changed until each of the input patterns has been presented to the nets and the corresponding delta-w's calculated. The presentation of all the input patterns is called an *epoch.* The delta-w's are accumulated at each link with each presentation, and the links are updated by the total after each epoch. (This is different from the standard algorithm, which updates the links after each presentation.)

The learning process continues until the net performs acceptably, or the net runs for a preset number of epochs, at which point the algorithm "breaks" to prevent wasting CPU cycles in an (evidently) infinite loop.

Using BackProp:

The Pre-Packaged System:

BackProp provides facilities that allow users to create and train system-defined networks with a minimum of effort. First-time users may find it helpful to work with one of these pre-defined networks first, to become comfortable with back-propagation, before moving on to creating and manipulating their own networks.

Using the pre-defined networks is simple. The user merely calls one of the functions that creates one of these nets: *SetUp424*, *SetUpXOR211*, or *SetUpXOR221*. These functions create the node and link objects of the net, and initialize the weights of the links with small random values as a symmetry breaking measure (as discussed in [PDP, pg. 330]). These functions also define the globals **inPatts** and **outPatts** to contain the input and corresponding output patterns on which the network is to be trained.

SetUp424 -- Creates a 4-2-4 network (a three layer network: four input nodes, two nodes in the hidden layer, and four output nodes. The layers are fully connected). **inPatts** and **outPatts** are set to the same list of patterns: {[1 0 0 0], [0 1 0 0], [0 0 1 0], [0 0 0 1]}. I.e., *Learn* will train the net to effect a subset of the identity relation.

SetUpXOR221 -- Creates a 2-2-1 network, also fully connected. **inPatts** is set to contain: {[0 0], [0 1], [1 0], [1 1]}, and **outPatts** contains: {[0], [1], [1], [0]}. I.e., *Learn* will train the net to effect the exclusive-or relation.

SetUpXOR211 -- Creates a 2-1-1 network, also fully connected. **inPatts** is set to contain: {[0 0], [0 1], [1 0], [1 1]}, and **outPatts** contains: {[0], [1], [1], [0]}. I.e., *Learn* will train the net to effect the exclusive-or relation.

To train the network, the user calls the *Learn* function (see below for a full

explanation of *Learn*). This function iteratively applies each of the input patterns to the input nodes of the net, forward-propagates this input signal to the output nodes, and then compares the output signal to the corresponding output pattern. If the output isn't acceptable, back-propagation learning takes place. When the network responds acceptably to all the input patterns, *Learn* exits.

During the learning process, *Learn* outputs diagnostic messages. The user controls the level of detail of these diagnostics (via the *SetVerbosity* function, explained below). The briefest diagnostics are of the form: "Epoch: <n> Error: <m> ", where <m> is a rough approximation of the sum of the error signals generated by each of the input patterns during the <n>th epoch. During training, <m> should tend toward 0.0, and should serve to give the user a rough indication of how well the training is progressing. (<m> is actually the sum of the squares of the differences between the actual and desired output values of each of the output nodes for all of the input/output patterns for which the network did not perform acceptably.)

When *Learn* exits, the user may test the network's training by using the *ShowBehavior* utility. This function applies each of the input patterns in **inPatts** to the net and compares the actual to the expected outputs. These results are printed in an easy-to-read table so that the user can see how well the net is performing for each of the input/output pattern pairs.

Monitoring the Net's Progress:

BackProp provides several utilities that allow the user to monitor the progress of the network as it trains. In order to use these utilities, though, the training process has to be suspended. *Learn* takes an optional key parameter, ":breakAt <n>", which if provided causes *Learn* to "break" every <n> epochs. (The Common Lisp *continue* function will restart the training at the point where the break occurred.) With the training suspended, the user may use the utilities (these are fully explained in the "Interesting Routines" section of this paper):

ShowBehavior -- As described above, this function prints a table detailing the current responses of the net to each of the input patterns, and compares these responses to the desired ones.

ShowLinks -- Prints a list of every link in the net, containing the current weighting of the links, and other information germane to links.

ShowNodes -- Prints a list of every node in the net. This function is mostly useful as the conjunct of *ShowLinks*, to see which links emanate from which nodes. The function prints the values of some of the fields of the node objects, but because these values are updated with each pattern presentation (unlike the links weights, which are updated only every epoch), the printed values are germane only to the most recently applied input pattern. *ShowBehavior* is a better means of evaluating the performance of the nodes.

Biasing

Use of *ShowLinks* yields output that might be confusing to first-time users. BackProp provides a "biasing" facility that is utilized by two of the *SetUp* functions. In the networks defined by these functions, each of the non-input nodes (nodes not in the input layer) is connected by an incoming link to a "bias" node. The bias node acts much like an extra input node whose output value is always 1.0. *ShowLinks*, then, will display more links than would exist in an unbiased net. For example, *SetUp424* will create a net with 11 nodes (one is the bias node) and 22 links: 8 between each layer, and 6 to the non-input nodes from the bias node. During training, the links from the bias node are altered like any others, incorporating the bias into the net's reactions. (The use of biasing is discussed some in [PDP]).

Appendix B contains an example session in which a network is created via *SetUp424*, and then trained.

Advanced Usage of BackProp

The *Learn* Function:

The *Learn* function provides several "key" parameters with which the user may alter the training process.

:**inPatts** inputPatterns

:**outPatts** outputPatterns -- The *SetUp* functions define the default input/output pairs in the globals **inPatts** and **outPatts**. The user may use these key parameters to supply different patterns on which to train the net. (Alternatively, the user could *setq *inPatts** and **outPatts**.) *InputPatterns* and *outputPatterns* should each be a list of patterns, where each pattern is a list of floating point values. The cardinality of the patterns in *inputPatterns* should be the same as number of nodes in the input layer of the net. The patterns in *outputPatterns* should be of a cardinality equal to the number of output nodes. *InputPatterns* and *outputPatterns* should be of the same cardinality.

:**learningRate** rate -- With this parameter the user may specify the learning rate constant ("h" in PDP). This value defaults to 0.3, but may be any value above 0.0.

:**momentum** momentum -- With this parameter the user may specify the momentum constant ("a" in PDP). This value defaults to 0.9, but may be any value above 0.0.

:**outputVerbosity** verbosity -- If *verbosity* = 1, diagnostic messages will be longer than if *verbosity*= 0, the default.

:**acceptableDiff** diff -- For any input pattern, if the activation level of each of the output nodes differs by no more than *diff* from its expected value, then the net's output is acceptable for that pattern and no learning occurs. This value defaults to 0.2.

:**breakAt** breakInterval -- If provided, causes *Learn* to enter a "break" loop every *breakInterval* epochs. The default is 0, meaning the loop will never break.

:**outputAt** epochs -- Diagnostic output detailing the effectiveness of the system will be printed every *epochs* epochs. The default is 20.

:**maxIters** limit -- *Learn* will break with an explanatory message after *limit* epochs. This is intended to prevent the training from infinite looping when the system "fails" to train in a "reasonable" amount of time. The value defaults to 5000 epochs--a value that is adequate for most small nets, providing *acceptableDiff* isn't too close to 0. If *limit* <= 0, "loop detection" is disabled.

Example (use of *Learn*):

```
(SetUp424)
(setq myInPatts '((0.0 0.0 0.0 0.0) (1.0 0.0 0.0 0.0) (1.0 1.0 0.0 0.0) (1.0 1.0 1.0 0.0)))
(setq myOutPatts '((0.0 0.0 0.0 0.0) (0.0 0.0 0.0 1.0) (0.0 0.0 1.0 0.0) (0.0 0.0 1.0 1.0)))
(Learn :inPatts myInPatts :outPatts myOutPatts :learningRate 0.2 :momentum 0.5 :acceptableDiff 0.25
    :breakAt 50 :infiniteLoop 1500)
```

In this example, the user wants to train a 4-2-4 network (with biasing) to effect a "counting" relation: the output of the net should be a binary representation of the number of input nodes that are "on". An output is considered acceptable if no output node's activation

level is more than 0.25 from its expected value. (I.e., for the second input/output pair, the output [0.1 -0.1 0.82 0.23] would be considered acceptable.) *Learn* will break every fifty epochs, allowing the user to examine the performance of the net at those times (probably via the *ShowBehavior* function). Lastly, the user expects the net will be trained within 1500 epochs.

Some of *Learn's* parameters also can be set at run time via the following set of access functions:

SetBreakCount [n] -- Sets to *n* the :breakAt parameter of *Learn*. *Learn* will break every *n* epochs. If no *n* is given, the call merely resets the *:breakAt* counter to the value previously set via *SetBreakCount* or the *:breakAt* parameter of *Learn*.

SetOutputCount [n] -- Sets to *n* the :outputAt parameter of *Learn*. *Learn* will print diagnostics every *n* epochs. If no *n* is given, the call merely resets the :outputAt counter to the value previously set via *SetOutputCount* or the *:outputAt* parameter of *Learn*.

SetMaxItersCount [n] -- Sets to *n* the :maxIters parameter of *Learn*. *Learn* will break after the *n*th epoch. If no *n* is given, the call is a no-op. If $n < 0$, loop detection is disabled.

SetVerbosity [n] -- If $n = 0$, subsequent diagnostics will be in a brief format. If $n = 1$, the diagnostics will be in a longer format.

The *Set..Count* calls are particularly useful for accessing BackProp's diagnostic routines during a learning session. For example, if the user was interested in the behavior of the learning algorithm after the 200th epoch, she might originally set :outputCount to 20, and :breakCount to 200. Then at the break on the 200th epoch, the user could use *SetOutputCount* to increase the frequency of output diagnostics.

Multiple learning session:

After *Learn* has successfully returned, the user may want to retrain the network, perhaps to compare the link weightings reached in different training sessions. Such comparisons are particularly interesting in network topologies having more than one minima in their "error surface".

To clear a net of the effects of a previous training, the user should call the *PurifyLinks* function (see description in "Interesting Routines", below). Then, to prepare the net for another training session, *Noise* should be called to initialize link weights to small random values, and thereby avoid a "symmetry breaking problem" [PDP, pg. 330].

Of course, the user doesn't have to use *PurifyLinks* and *Noise* between sessions; given enough epochs, the learning process will eventually overcome any initial weighting. However, the training time for a network increases as the initial weighting becomes more extreme.

Constructing your own nets (for fun & profit!)

The easiest way to explain how to create nets is to explain the workings of one of the *SetUp* functions. Explained in detail below, is the code for *SetUpXOR*.

```
(defun SetUpXOR221
  (&aux in hid out)                ; Tmp variables holding the nodes of the
                                   ; input, output, & hidden layers of the net.

"Creates a 2-2-1 network, intended for learning the XOR encoding:
00-->0, 11-->0, 01-->1, and 10-->1.
These input and output patterns are stored in '*inPatts*' and '*outPatts*'."

  (setq *inPatts* '((0.0 0.0) (1.0 1.0) (0.0 1.0) (1.0 0.0)))
  (setq *outPatts* '((0.0) (0.0) (1.0) (1.0)))

  (ClearNet)
  (setq in (DefGroup 2 t))
  (setq hid (DefGroup 2))
  (setq out (DefGroup 1))
  (InterConnect in hid)
  (InterConnect hid out)

  (Noise .2)
  (setq *levels* (list in hid out))
  (quote 'XOR-2-2-1-Net))
```

(ClearNet) -- This call erases any existing nodes and links from previously defined networks. The space occupied by the deleted objects is returned to the free heap. ***ClearNet* should always be called before creating a new net.**

(setq *inPatts*....) (setq *outPatts* ....) -- The default input and output patterns are defined.

(setq in (DefGroup....)) , etc. -- *in*, *hid*, and *out* are holding variables for the nodes of the input, hidden, and output layers of the network being defined. *DefGroup* returns a list of the specified number of nodes. The second parameter specifies whether the nodes should be biased. If the second parameter is "nil" (the default), then links are created between the bias node and the returned nodes. These links are appended to the global **links**.

(InterConnect ....) -- This call creates links fully connecting the given two sets of nodes. In this example, adjacent levels are fully connected. This is the primary means of creating links in the net. The links are appended to the global **links**.

(Noise ...) -- This function effects the symmetry-breaking strategy outlined in [PDP, pg. 330]. The weight of every link in the net (i.e., all the elements of **links**) is initialized to a random number in the given range.

(setq *levels* ...) -- Lastly, the global **levels** **must** be a list of the levels of the network. Each "level" is in turn a list of node objects. ***Note!*** *The input layer* ***must*** *be the first element of* *levels*, *and the output layer must be the last element.*

Provided all goes well, a network should now exist, ready for a call to *Learn*. The globals **links** and **levels** are referenced throughout the program, so it is important that they be properly initialized. Using the functions described guarantees this.

Another example (creating a different XOR net):

A different kind of XOR net is described in [PDP, pp. 331-335] and it is illustrative to describe how such a net can be created using the methods outlined above. The net is a modified 2-1-1, but with the two input nodes connecting to both the hidden layer and the output layer (each with only one node). Here we give the commented code for *SetUpXOR211*, a pre-defined function that creates such a network. Differences between the 2-1-1 and the 2-2-1 XOR nets are noted in the function's comments.

```
(defun SetUpXOR211
  (&aux in hid out)                ; Tmp variables holding the nodes of the
                                   ; input, output, & hidden layers of the net.

"Creates an unbiased 2-1-1 network, intended for learning the XOR encoding:
00-->0, 11-->0, 01-->1, and 10-->1.
These input and output patterns are stored in '*inPatts*' and '*outPatts*'.
The input layer is connected to both the hidden and output layers."

   (setq *inPatts* '((0.0 0.0) (1.0 1.0) (0.0 1.0) (1.0 0.0)))
   (setq *outPatts* '((0.0) (0.0) (1.0) (1.0)))

   (ClearNet)
   (setq in (DefGroup 2 t))
   (setq hid (DefGroup 1 t))          ;Note that we aren't biasing the non-input nodes.
   (setq out (DefGroup 1 t))
   (InterConnect in hid)    ; Just like 2-2-1, but...
   (InterConnect in out)    ;The input layer is also connected directly
                            ;to the output layer.
   (InterConnect hid out)

   (Noise .2)
   (setq *levels* (list in hid out))
   (quote 'XOR-2-1-1-Net))
```

Implementation Notes:

Memory Usage & Garbage Collection:

BackProp was developed on a Tektronix 4405 with a relatively small amount of heap space (~2.5 Megs), using Tektronix's implementation of Common Lisp. Early on it was noticed that during training the machine seemed to do garbage collections rather frequently. Analysis of this occurrence revealed that in the implementation of Common Lisp that we were using, the floating-point operations allocated memory from the free heap upon every invocation. This was true regardless of whether the program was compiled or interpreted. For example, each floating-point multiplication used eight bytes of heap space! This problem has existed on all the machines the authors have yet run this program on, including TI Explorers, and Sun 3.0's running Kyoto Lisp.

With even small nets, such as the ones defined by the *SetUp* functions, the number of numerical computations executed in a training session is quite high. Thus, a warning is in order: if your implementation of Common Lisp allocates heap space to effect floating-point calculations, you should expect fairly frequent garbage collection (how "frequent" would depend on the size of your heap, of course). If the reader wishes to avoid this problem, the authors suggest altering the code so as to use scaled bignums instead of floating-point values. This is really nothing but a poor man's implementation of fixed-point technology. On our machines, though, the fixed-point calculations had the same memory-chomping problem as the floating-point, while the bignum calculations did not.

For example, a bignum version of BackProp might scale all floating-point values by 1000, so that 10.015 became 10015, etc. This scheme might result in some loss of precision, especially for floating-point values particularly near to, or far from zero. If the user doesn't mind the garbage collections, then these steps obviously aren't necessary.

Coding Style

Naming Conventions:

The case of the characters in symbol names is significant. The first character of function names are always capitalized. The first character of all other symbol names are lower case.

In most symbol names, capitalization is used to delineate word boundaries. Occasionally, an underbar character is used. For example, "aVariableName" versus "a_variable_name".

Globals are identified by bracketing asterices, or a "g" prefix. For example, "*links*" or "gBPOutputCount". (The "*. . .*" form is reserved for truly application-wide globals, whereas the "g-" form is typically used for symbols that would better be scoped to just a small set of routines. Lisp is somewhat lacking in such scoping capabilities, however.)

The names of constants (defined via *defconstant*) are prefixed with a "c". For example, "cBPBreakCount".

Documentation:

The "header comments" for each function are given in their *documentation field* as provided by Common Lisp. This documentation may be referenced at run-time by the *doc* function. (Some implementations of Common Lisp have built-in methods for accessing documentation fields.) Usage of *doc* is described below, in the "Important Routines" section of this paper. For most BackProp symbols, "(doc <symbolName>)" will suffice. For example: "(doc 'doc)" will print a description of *doc*.

Important Routines of BackProp:

The following function descriptions are intended to be brief. Fuller descriptions of most of the routines can be found in their "documentation" field.

MakeSigmoid -- Creates a set of look-up tables used to approximate a sigmoid function.

Sigmoid value -- This routine acts in concert with *SetInput* to effect the activation function of the network. Returns the (approximated) value of the standard sigmoid function for the given *value*. The accuracy of the function is greatest for *values* around 0.0.

SetInput node -- Sets the *input* field of the given node object, *node*, to the sum of the weighted outputs of the nodes inputting to *node*. During a forward propagation pass, the *output* field of every non-input node, *i*, is set to *Sigmoid(SetInput(i))*.

DefGroup numNodes [unbiased-p] -- Returns a list of *numNodes* newly generated node objects. If *unbiased-p* is nil (the default), this function will create link objects connecting the bias node and the returned node objects. These links are stored in **links**.

InterConnect fromNodes toNodes -- Creates and adds to **links** link objects connecting every node object in the list of nodes *fromNodes* to those in *toNodes* (typically, created by *DefGroup*).

ClearNet -- The **links** and **levels** globals are cleared and other actions are taken to insure that the memory allocated to the existing links and nodes will be freed during the next garbage collection.

Noise [maxNoise] -- Sets the weight of every link in **links** to a random value between *-maxNoise* and *maxNoise*. If no parameter is given, *maxNoise* defaults to 0.2. This function should be called before every learning session.

PurifyLinks -- Clears in every link object in **links** all fields that affect the learning process of a training session. (I.e., this routine does not clear fields used to store temporary values.) Should be called between training sessions of a network, usually followed by a call to *Noise*.

AcceptablePerformance nodes acceptable-diff -- Returns *true* if the error signals of the list of node objects, *nodes*, is acceptable.

BackProp node -- Propagates the error signal of the given node to the nodes that input to *node*.

ForwardPass inputs -- Effects a forward-propagation pass through the network. The activation levels of the nodes in the input layer are locked to the values in the list of floating-point numbers, *inputs*. Then the standard forward pass is executed, propagating activation levels to the output nodes. The cardinality of *inputs* should equal the number of input nodes in the network.

BackwardPass desired-output acceptable-diff -- Effects a backward-propagation pass through the network, but doesn't effect actual training (*UpdateWeights* does that). The activation (output) levels of the nodes in the output layer of the network are compared against the corresponding values in the list *desired-output*. If an output node's activation level is less than *acceptable-diff*, then that node's error signal is zero. Unless all the output nodes have zero error signal, the error signals are propagated backwards through the net in the manner described above in "Description of Back Propagation".

UpdateWeights learningRate momentum -- This routine applies to the weight of every link in **links** the "delta-w's" (changes to link weights) calculated by *BackwardPass* and accumulated (in the *delta* field of each link object) during an epoch. For each link, new weight -= (*learningRate* * <summed deltas>) + (*momentum* * <prev delta>), where <prev delta> was the link's delta-w at the previous call to *UpdateWeights*.

Learn [**:inPatts** inPatts] [**:outPatts** outPatts] [**:learningRate** l] [**:momentum** m] [**:acceptableDiff** d] [**:outputVerbosity** v] [**:breakAt** b] [**:outputAt** o] [**:maxIters** z] -- This function is described in detail in "Advanced Usage", above.

Doc symbolName [symbolType] -- Returns the documentation field associated with the given symbol, *symbolName*. *SymbolType* may be any of the values accepted by the Common Lisp function *documentation*. Standard values are 'function, 'structure, 'variable, and 'constant. If *symbolType* is not given, *Doc* first searches for documentation for a function of the given name, then a "struct", then a "defvar"–ed symbol, and lastly a "defconstant"-ed symbol. If no such symbol is found, returns nil.

OutputToFile file -- Opens for writing the file named *file*. If no such file exists, one is created. Any existing data in the file will be overwritten. All subsequent output will be directed to that file. A useful function for those of us who work on systems where output cannot be redirected via a system call!

OutputToScreen -- Subsequent output will be directed to the screen. If output was previously directed to a file via the *OutputToFile* command, this command closes that file.

FindLink aLinkName -- Returns the link object with the given name, *aLinkName*. If no such link exists in **links**, the function returns nil. Link names are displayed by the *ShowLinks* function.

FindNode aNodeName -- Similar to *FindLink*. Returns the node object corresponding to the given name. If no such object exists in **levels**, the function returns nil. Node names are displayed by the *ShowNodes* function.

ShowLinks -- Prints a listing of all the link objects in the net and the values of their various fields.

ShowLink link -- Prints the current values of the fields of the given link object, *link*.

ShowNodes [verbose-p] -- Prints a listing of all the node objects in the net (except the bias node), and the current contents of their fields.

ShowNode node [verbose-p] -- Prints the current values of the fields of the given node object, *node*. If *verbose-p* is non-nil, gives a more detailed description.

SetUp424 -- Erases any existing network and creates a 4-2-4 network. Defines **inPatts** and **outPatts**. Described in detail in "The Prepackaged System", above.

SetUpXOR221 -- Erases any existing network and creates a 2-2-1 network. Defines **inPatts** and **outPatts**. Described in detail in "The Prepackaged System", above.

SetUpXOR211 -- Erases any existing network and creates a 2-1-1, non-standard network. Defines **inPatts** and **outPatts**. Described in detail in "The Prepackaged System", above.

ShowBehavior [**:acceptable-diff** n] [**:inputs** in] [**:outputs** out] [**:verbose-p** v] -- Prints diagnostics on the performance of the network. Each element of the list *in* should be a list of *j* numeric values, where *j* is the number of nodes in the input layer. *Out* should be of the same format, but where *j* is the number of output nodes. *N* is the acceptable difference between the actual and desired (those in *out*) values of the activation levels of the output nodes. If *v* is non-nil, the diagnostics will be in a longer format. *In* defaults to **inPatts**, *out* to **outPatts**, *n* to *gBPAcceptableDiff* (set by the *:acceptableDiff* parameter to *Learn*), *v* to true.

For each element of *in* (*out* should be of the same cardinality) *ShowBehavior* locks the activation levels of the net's input nodes to the values of the element (a list of numeric values). The activation levels are forward-propagated to the output nodes, and compared against the values of the corresponding element of *out*. *ShowBehavior* prints the inputs, the desired and actual activation levels and the error signals of the output nodes.

SetBreakCount [iters] -- Sets the "break counter" to *iters*. This counter is decremented with each epoch. When the counter reaches 0, *Learn* breaks. Upon a *continue* the counter is reset to *iters*. Thus, *Learn* will break every *iters* epochs. If *iters* < 0, *Learn* never breaks (except, perhaps, for the action of *SetMaxItersCount*). If *iters* is not given, the counter is reset to the most recent value explicitly given via *SetBreakCount*, or via the *:breakAt* key parameter of *Learn*.

SetOutputCount [iters] -- Sets the "output counter" to *iters*. The operation is identical to *SetBreakCount*. When the counter reaches 0, *Learn* prints diagnostics. The level of detail of these is controlled by *SetVerbosity*.

SetMaxItersCount [iters] -- Sets the limit on the number of epochs in current learning session to *iters*. Upon finishing the *iters*-th epoch, *Learn* will break with the message, "BREAKing because of too many epochs. (Non-stabilizing net?)" If *iters* < 0, *Learn* disables loop detection. If *iters* is not given, the call has no effect.

Work in Progress....

Currently, we are writing code that will allow users to define their own activation functions for the network. The authors feel that users would find it very informative to be able to train the same network on the same inputs but using different activation functions. (Some examples can be found in [PDP, Chapter 10].)

By next Fall, the authors hope to have a graphic-oriented, tutorial version of this package running on Apple's Macintosh II. As planned, the system will utilize the strong capabilities of the Mac II's color graphics, and will be implemented in Allegro's Coral Common Lisp. Such a system should be ideal for teaching some of the basic concepts of connectionist theory.

References

Rumelhart, D.E., & McClelland, J.L. (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations.* Cambridge, MA: MIT Press.

Jones, W.P., & Hoskins, J. Back-Propagation, a Generalized Delta Learning Rule. *Byte*, October, pp. 155-162.

Appendix A: The code[1]

```
(defconstant cBPLearningRate .3)
(defconstant cBPMomentum .9)
(defconstant cBPAcceptableDiff .2)
(defconstant cBPOutputCount 10)
(defconstant cBPBreakCount 0)
(defconstant cBPMaxIterCount 10000)


(defvar gCurNode 0              "'Name' of the last node created. See BPNode.")
(defvar gCurLink 0              "'Name' of the last link created. See BPLink.")


(defvar gBPBreakCount cBPBreakCount
   "BP will break every <> epochs.  See SetBreakCount.")
(defvar gBPOutputCount cBPOutputCount
   "BP will output diagnostics every <> epochs.  See SetOutputCount.")
(defvar gBPMaxIterCount cBPMaxIterCount
   "BP will break on the <>th epoch, as prevention against infinite looping.")
(defvar gBPIterCntr 0 "Number of epochs so far executed.")
(defvar gBPOutputCntr gBPOutputCount "Counts epochs for <gBPOutputCount>.")
(defvar gBPBreakCntr gBPBreakCount "Counts epochs for <gBPBreakCount>.")


(defvar gBPVerbosity 0
   "Controls verbosity of output messages. Can be 0,1. See CheckEpochCounts.")
(defvar gBPLearningRate cBPLearningRate
   "Learning rate of the net.")
(defvar gBPMomentum cBPMomentum
   "Momentum factor of learning alg for net.")
(defvar gBPAcceptableDiff cBPAcceptableDiff
   "Response of output node is acceptable if within this amount of the
   desired response.")


(defvar *stdio* t "Most Format calls direct their output to this stream.  If
value is 't', output goes to StdIO.")

(defstruct (BPLink)
   "Represents a link between two nodes in the network. Contains the
   weight of the link, the delta from the last time the weight was updated,
   etc.
   See BPNode."

  (fromNode    nil)              ; Link goes from this node to...
  (toNode      nil)              ; ...this node.
  (weight      0 :type short float)
  (delta       0 :type short float); Amount the weight will be changed during
                                  ; this pass.
  (prevDelta   0 :type short float); Amount the wght changed during prev pass.
```

1 (c) Pollack, Evett, and Hendler, March, 1988.
For more information about BackProp contact Matt Evett (evett@mimsy.umd.edu) or Jim Hendler (hendler@mimsy.umd.edu), Dept. of Computer Science, University of Maryland.

```
  (name                          ; Every time a BPLink is created, gCurNode
                                 ; will be incremented, so each link will have
                                 ; a unique name.
     (setq gCurLink (+ 1 gCurLink))))

(defstruct (BPNode)
 "The other major data structure in BackProp (see BPLink for other). These
 objects represent the nodes of the net (including the bias node).
 See BPLink."

  (inLinks       nil)
  (outLinks      nil)
  (input         0.0 :type short float)
  (output        0.0 :type short-float)
  (errSig        0.0 :type short float)
  (incomingErrSig      0.0 :type short float)
  (name
     (setq gCurNode (+ 1 gCurNode))))


(defvar *links* nil "List of the link objects comprising the network")

(defvar *levels* nil              "Each elem of this list is a list of nodes
                                    a group    comprising one level of the net.")
(defvar *biasNode* nil  "The weights of the links from this node to the
                         input nodes are added to the input of those nodes.")

(defvar *inPatts* nil "List of input patterns for the net.")


(defvar *outPatts* nil "List of output patterns corresponding to *inPatts*.
When an input pattern is asserted on the net's input nodes, the net's output
should be within an acceptable range of the corresponding output pattern.")


(defvar cSig-table-coarse-size 200)
  "The number of elements of the coarse Sigmoid array for each of the pos &
  neg vals defined by the table.  I.e., 'coarse' table will use 'size' elts
  for the (0,max] range, and 'size' for [ max,0), and one elt for the val of
  the Sigmoid fnct at 0.0."

(defvar cSig-table-coarse-max 20.0)
(defvar cSig-table-coarse-scale
   (/ cSig-table-coarse-max cSig-table-coarse-size))

(defvar cSig-table-fine-size 250)
(defvar cSig-table-fine-max 5.0)
(defvar cSig-table-fine-scale
   (/ cSig-table-fine-max cSig-table-fine-size))

(defvar cSig-table-extra-fine-size 100)
(defvar cSig-table-extra-fine-max 1.0)
(defvar cSig-table-extra-fine-scale
   (/ cSig-table-extra-fine-max cSig-table-extra-fine-size))

(defvar gSigCoarseArray nil)   ; The look up tables for the Sigmoid function.
(defvar gSigFineArray nil)     ; See Sigmoid.
(defvar gSigExtraFineArray nil)
```

```
(defun MakeSigmoid ()

  "Creates the sigmoid lookup tables, using the cSig- table consts.
   See: CreateSigLookup, Sigmoid"

   (setq gSigCoarseArray
         (CreateSigLookup
            (* 2 cSig-table coarse size) cSig table coarse max))
   (setq gSigFineArray
         (CreateSigLookup
            (* 2 cSig table fine- size) cSig table-fine max))
   (setq gSigExtraFineArray
         (CreateSigLookup
            (* 2 cSig table extra-fine size) cSig-table extra-fine-max))
)


(defun CreateSigLookup

   (size                  ; Determines the granularity of the array.
                          ; This will be the # of elems in the array.
    maxVal                ; ( maxVal, maxVal) is the "domain" of the
                          ; table.

   &aux
   sigmoidTable)          ; The table we're creating.

   "Creates a look up table of the values of a 'sigmoid' function. The
  table's granularity is specified implicitly by the given cardinality of
  the table and the min and max domain values the table is to index. The
  0th element of the array will be the val of the sigmoid function at
  < maxVal>, and the <size> 1th element will be the val at <maxVal>."

  (setq sigmoidTable (make array (+ size 1) :element type 'single-float))
  (do
     ((curElem 0 (+ curElem 1))
      (curDomainVal
         (coerce ( maxVal) 'single float)
         (+ curDomainVal increment))
      (increment (/ (* 2.0 maxVal) size)))

     ((> curElem size)          ; Loop for each elem of the table...
      sigmoidTable)

     (setf (aref sigmoidTable curElem) ; Set each table elem to val of sigmoid
                                ; function
       (/ 1.0
          (+ 1.0
             (exp ( curDomainVal)))))
  ))

(defun Sigmoid
    (x)                         ; The val we're to eval the sigmoid fnct on.
    "Returns an approximation of the value of the sigmoid function for a
     given value. The sigmoid function is approximated with a table lookup.
     The lookup is actually done on one of three tables: A 'coarse' table is
     used for the values fairly far from the cross over point of the sigmoid
     function, where the slope of the function is gentle. A 'fine' table is
     used for the values near the cross over, and an 'extrafine' table is
```

```
      used for those values very near the cross over, where the slope of the
      function is quite steep.  See MakeSigmoid."

    (cond
       ((> x cSig-table coarse max) 1.0); For distant vals, rtn asymptotes
       ((< x (  cSig-table-coarse max)) 0.0)

       ((> (abs x) cSig table-fine max); Fairly distant vals ref coarse tbl
        (elt gSigCoarseArray
             (+ cSig-table-coarse size
                (round (/ x cSig-table coarse scale)))))

       ((> (abs x) cSig-table extra-fine max); Nearer vals
        (elt gSigFineArray
             (+ cSig-table-fine size
                (round (/ x cSig-table fine scale)))))

       (t                              ; Nearest vals
        (elt gSigExtraFineArray
             (+ cSig-table extra fine-size
                (round (/ x cSig table extra fine scale)))))
       )
    )

(defun DefGroup
   (size                    ; The # of nodes in the group.
   &optional
   (in-input-levelP nil) ; T      group's nodes are input nodes.
   &aux
   (the-new group '()))          ; The newly created group of nodes.

  "Creates a group of nodes and returns them in a list.
  SIDE EFFECT: If the group is to be a 'non input' group, then the function
  will create links (and so added to *links*) from *biasNode* to the
  nodes of the new group.  It might be better to move this functionality
  elsewhere, perhaps into its own function...."

  (dotimes (numCreated size)
        (setq the new group (cons (make BPNode) the new group)))

  (if (not in input levelP)       ; SIDE EFFECT!!!  If the group is not in
                                  ; the input level, then we create links from
                                  ; *biasNode* to the nodes of the group.
     (InterConnect (list *biasNode*) the new group))

  (return from
     DefGroup the new group); Result of this function is the new group

)


(defun InterConnect
   (from group                    ; Links are created from every node in this
                                  ; group to every node in....
    to group)                     ; ...this group.

  "Fully links the given groups of nodes.  Links are FROM the first group's
  nodes TO the second group's nodes.
  SIDE EFFECT: Note that this function alters *links* and several fields of
```

```
  the nodes of the given groups."

  (dolist (from_node from_group)
          (dolist (to-node  to_group)
               (AddLink from_node to_node)
          )))

(defun AddLink
   (from_node                    ; Link will go from this node to...
    to-node                      ; ...this node.
    &aux
    the-new-link)                ; The link created.

  "AddLink(fromNode, toNode) creates a link between the two nodes and adds
  it to *links*.  Also alters the outLinks field of fromNode and the
  inLinks field of toNode to contain the newly created link.  The function
  returns the newly created link."

  (setq the-new-link
        (make-BPLink :fromNode from_node :toNode to_node))
  (push the-new-link *links*)
  (push the-new-link (BPNode-outLinks from_node))
  (push the_new-link (BPNode_inLinks to_node))

  (format *stdio* "created link <~d> from [~d] to [~d]~%"
          gCurLink (BPNode_name from_node) (BPNode_name to_node))

  the-new-link
)

(defun ClearNet ()

  "Disconnects all links (so gc will work properly on them) and nodes, and
  resets the *biasNode* to have no links."

  (dolist (x *links*)
          (setf (BPLink_fromNode x)
               (setf (BPLink_toNode x) nil)))
  (dolist (group *levels*)
          (dolist (node group)
                  (setf (BPNode-inLinks node)
                       (setf (BPNode_outLinks node) nil))))
  (setq *links* nil)
  (setq *levels* nil)
  (setq *biasNode* (make_BPNode :output 1.0))
)

(defun Noise
   (maxNoise)                    ; Noise is in range (-maxNoise,maxNoise).

  "Sets the weight field of each link in the net to a randomly generated
  value between  <parm> and +<parm>."

; Actually the random vals are in the range (-maxNoise,maxNoise), but who's
; counting?

  (dolist (link *links*)
          (setf (BPLink_weight link)
               (-  (random (* 2.0 maxNoise)) maxNoise))))
```

```
(defun PurifyLinks ()
  "Clears the fields of all link objects."
   (dolist (link *links*)
          (setf (BPLink-weight link) 0.0)
          (setf (BPLink-prevDelta link) 0.0)
          (setf (BPLink-delta link) 0.0)))

(defun SetInput
   (node                          ; We want to set the input of this node.
   &aux
   (cur-input 0.0))

   "Sets the input of a given node to the sum of the outputs across the
   incoming links to the node. The output of a link = weight of the link
   * the output of the node at the other end of the link. The function
   returns the calculated input value."

   (dolist (input-link (BPNode-inLinks node))
          (incf cur-input
                (* (BPLink-weight input-link)
                   (BPNode-output (BPLink-fromNode input-link)))))
   (setf (BPNode-input node) cur-input))

(defun AcceptablePerformance
   (group                         ; Group of nodes, typically an output layer.
    acceptable-diff)              ; Acceptable error signal.
"Returns 't' if the error signal of every node in the given list of nodes is
less than the given 'acceptable-diff' value. Typically the function is
called on the output layer of the net to determine if the output values of
the net are all within an acceptable limit of the desired output pattern.
SEE BackwardPass."

    (dolist (node group)
          (if (> (abs (BPNode-incomingErrSig node)) acceptable-diff)
             (return-from AcceptablePerformance nil)))
    t)

(defun BackProp
   (node)                         ; The delta of this node is to be propogated
                                  ; to the nodes inputting to it.
"Propogates a given node's incoming error signal backwards to its incoming
nodes. The function first computes the error signal of the node using the
relation: errSig = incomingErrSig * (output * (1 - output)), where
incomingErrSig = {sum of error sigs of the nodes to which this node outputs}.
The error signal is propogated to the node's incoming links and nodes thus:
   The errorSignal ('delta w' in PDP) of each incoming link gets the node's
   error signal, weighted by the node's activ. value. (Thus the delta w's are
   only accumulated during a epoch. The links' weights themselves aren't
   changed until after the epoch is completed).
   The error signal is also propogated to inputting nodes. The node's error
   signal is accumulated in the 'incoming error signal' field of each
   inputting node. The signal is weighted by the weight of the connecting
   link."

   (setf (BPNode-errSig node)  ; errSig = incomingErrSig * (output * (1 - output))
         (* (BPNode-incomingErrSig node)
            (BPNode-output node)
            (- 1.0 (BPNode-output node))))
```

```
  (dolist (in_link (BPNode-inLinks node)) ; Propogate delta to each
                                          ; inputting node and link.
          (incf (BPLink-delta in-link)
                (* (BPNode-errSig node)
                   (BPNode-output (BPLink-fromNode in_link))))
          (incf (BPNode-incomingErrSig (BPLink-fromNode in_link))
                (* (BPNode-errSig node)
                   (BPLink-weight in-link)))))

(defun ForwardPass
   (inputs)                          ; The outputs of the input nodes will be
                                     ; bound to these values.
   "Effects one forward pass through the network.
   First, the input nodes' outputs
   are set to the values given in <parm>, which should be a list of fixed
   floating point numbers, one for each input node (the nodes of the 1st
   group in *levels*, usually the nodes to which the *biasNode* is linked).
   The the output values are propogated through the network,
   level by level. The function returns as a list the output values of the
   nodes in the final level."

   (dolist (group *levels*)          ; Set Input field of every node to 0.0.
          (dolist (node group)
                  (setf (BPNode-input node) 0.0)))

 ; Set Input fields of every 'biased' node to the 'weight' of the link from the
 ; *biasNode* (i.e., activation value of the biasNode is 1.0). I.e., the
 ; inputs of these nodes are "biased".

   (dolist (link (BPNode-outLinks *biasNode*))
          (incf (BPNode-input
                   (BPLink-toNode link))
                (BPLink-weight link)))

   (LockOutputs (car *levels*) inputs)  ; Set the input nodes' output fields
                                        ; to the values specified in parm .

 ; Now we actually do the propogation. We work through the net level by
 ; level. For each node in a level we set its input to the sum of the
 ; outputs across the incoming links. Then we set its output to be
 ; Sigmoid(input).

   (dolist (level (cdr *levels*))
          (dolist (node level)
                  (setf (BPNode-output node)
                        (Sigmoid (SetInput node)))))

   (mapcar #'BPNode-output    ; Forms a list of the output fields of the
                              ; nodes of the output level, & RETURNS it.
           (car (last *levels*))))


(defun BackwardPass
   (desired-output                  ; The desired output pattern.
      acceptable-diff               ; If each output differs from its desired value
                                    ; by no more than this amount, then no
                                    ; learning will be done (0.0 returned).
      &aux
```

```
    levels-r                    ; A reversed list of the net's levels.
    (squared-error 0))          ; Sum of the squares of the deltas of the
                                ; output nodes.
```

"Effects the backward pass phase of back propogation. Starts at the output level and calculates the difference between the desired and actual output values of every node in the output level. These 'deltas' are then propogated backwards through the net. The function returns the sum of the squares of the output nodes' deltas."

```
  (setq levels-r (reverse *levels*)) ; So we can go from output -> input...

  (dolist                   ; Clear the incomingErrSig fields of all nodes.
                            ; This is necessary because in hidden units
                            ; this val is incremented, not setq'd.
     (level *levels*)
     (dolist
        (node level)
        (setf (BPNode-incomingErrSig node) 0.0)))

; For each output node, set its incomingErrSig field to the difference between the
; output node's activation level and the desired level (as specified in the
; desired-output parm).

  (setq squared-error
        (OutputResponse (car levels-r) desired-output)) ; Sets .incomingErrSig's.

  (if (AcceptablePerformance (car levels-r) acceptable-diff)
     (return-from BackwardPass 0.0))   ; If delta is acceptable, leave.

  (dolist (level levels-r) ; Level by level, propogate the deltas
                           ; backward at each node in the level.
          (dolist (node level)
                  (BackProp node)))

  squared-error) ; RETURN sum of the squares of the deltas of the output nodes.


(defun Learn
   (&key
    (inPatts *inPatts*)          ; List of patterns we want the net to
                                 ; correlate to....
    (outPatts *outPatts*)        ; ...these output patterns.

    (learningRate gBPLearningRate); "mu". Degree to which weights are
                                 ; affected by their calc'd error signal.
    (momentum gBPMomentum)       ; "alpha". Degree to which link weights
                                 ; will continue to change by amount of
                                 ; previous changes.
    (acceptableDiff gBPAcceptableDiff) ; Learning is done if net's outputs from
                                 ; inPatts correspond to desired
                                 ; outPatts w/in this amount.
    (outputVerbosity gBPVerbosity)
                                 ; Length of output msgs: 0 = short, 1 = long
    (breakAt gBPBreakCount) ; Breaks every breakAt iterations.
    (outputAt gBPOutputCount)    ; Prints diagnostic output every - iters.
    (maxIters gBPMaxIterCount))  ; After this many epochs of the learning
                                 ; algorithm, the will terminate. Learning is
                                 ; easily continued by calling Learn again...
```

```
"Successively applies each of the input patterns to the net and does a
 Forward Pass to generate an output pattern, then does a backward pass to
 propogate the error signal across the network.  During backward passes, the
 links accumulate their deltas.  Only after all the inputs have been applied
 does the function update the links' weights by their accumulated deltas.  The
 routine returns if the net reacts within the 'acceptable difference' for
 each input pattern."


  (if (not gSigCoarseArray)              ; Sigmoid look-up tables haven't yet
                                         ; been defined...
    (MakeSigmoid))                       ; ...then define them.

  (setq gBPAcceptableDiff acceptableDiff); These might better be fncts.???
  (setq gBPMomentum momentum)
  (setq gBPLearningRate learningRate)

  (SetVerbosity outputVerbosity)

  (SetBreakCount breakAt)    ; See CheckEpochCounts....
  (SetOutputCount outputAt)
  (SetMaxItersCount maxIters)
  (ResetEpochCount)

  (do ((epoch 0 (+ 1 epoch))    ; Loop until net behavior is acceptable.
       (error  1.0)             ; Sum of errors of net for all input pats.
       (breakIters 0 (+ 1 breakIters)); Counter for  breakAt .
       (outputIters 0 (+ 1 outputIters))); Counter for  outputAt .
      ((= 0.0 error) 'whew)

      (dolist (link *links*)     ; Clear .delta fields of links.
              (setf (BPLink  delta link) 0.0))

     (setq error 0.0)    ; Clear error accumulator.
     (do
        ((remainingInPatts inPatts (cdr remainingInPatts))
         (remainingOutPatts outPatts (cdr remainingOutPatts)))
        ((not (and remainingInPatts remainingOutPatts)) t)

       (ForwardPass (car remainingInPatts)) ; Run on an input.
       (incf error
             (BackwardPass (car remainingOutPatts) acceptableDiff)))

; At this point, we have accumulated the error signals from each of the
; input/output pattern pairs.  Also, the .delta fields of the links are the
; accumulation of the deltas calculated for each such pair.  We give the
; user a chance to view the net, now, before the link weights are updated by
; their accumulated deltas

      (UpdateWeights learningRate momentum)

      (CheckEpochCounts error)

      )
   (format *stdio* "Acceptable net successfully created!  Epoch: ~d~%"
           gBPIterCntr))

(defun UpdateWeights
```

```
    (learningRate                    ; See Learn for details.
     momentum)

"Updates the weights of all the links.  A weight's change is proportional to
 the sum of the deltas calculated during one epoch through all the input
 patterns, and to the change made to that weight in the previous epoch.  The
 full equation:
     weight  = (learningRate * summedDeltas) + (momentum * prevDelta)
 LearningRate and Momentum are referred to as 'mu' and 'alpha' in PDP."

    (dolist (link *links*)
            (incf (BPLink  weight link)
                  (-
                     (setf (BPLink-prevDelta link)
                           (+
                              (* learningRate (BPLink  delta link))
                              (* momentum (BPLink  prevDelta link))))))))



(defun LockOutputs
    (nodesToLock                          ; List of nodes whose outputs are to be
                                          ; locked
     outputVals)                          ; ...to these values.

"Sets the output field of each of the given list of nodes to the
corresponding value in the given list <parm2>, which perforce should be
the same length as <parm1>."

    (do ((nodes nodesToLock (cdr nodes))
         (node)
         (remaining outputVals (cdr remaining))
         (outputVal))
        ((not (and nodes remaining))
         nil)

        (setq node (car nodes))
        (setq outputVal (car remaining))

       (setf (BPNode  output node) outputVal))
)



(defun SimpleNet
   (&aux
    in hid out)              ;Temporaries.
    (ClearNet)
    (setq in (DefGroup 2 t))
    (setq hid (DefGroup 3))
    (setq out (DefGroup 1))
    (InterConnect in hid)
    (InterConnect hid out)
    (setq *levels* (list in hid out))
    (PurifyLinks)
    (print "A simple 2  3  1 net created."))

(defun DirectOutput (fileName)
```

```
  (setq *stdio*
        (open fileName :direction :output
              :if exists :overwrite
              :if does not exist :create)))

(defun OutputToFile (fileName)
 "Subsequent output will be directed to the file with the given name. If such
 a file doesn't exist, it will be created. The contents of an existing file
 will be erased. See OutputToScreen."

  (DirectOutput fileName))

(defun OutputToScreen ()
 "Subsequent output will be directed to the screen (stdio)."
   (if (streamp *stdio*)
       (close *stdio*))
   (setq *stdio* t))

(defun FindLink (aLinkName)
 "Returns the link object having the given name."

  (do ((curList *links* (cdr curList)))
      ((not curList) nil)
      (if (= aLinkName (BPLink-name (car curList)))
          (return (car curList)))))

(defun FindNode (aNodeName)
 "Returns the node object having the given name.
 See ShowNode, ShowNodes."

   (dolist (curGroup *levels* nil)
           (dolist (curNode curGroup)
                   (if (= aNodeName (BPNode-name curNode))
                       (return-from FindNode curNode)))))

(defun ShowLinks ()
 "Outputs a listing of all the links in the net. This includes any biasing
 links.
 See ShowLink."

  (format *stdio* "Current contents of *links*:~%")
  (dolist (x *links*)
          (ShowLink x))
  (terpri *stdio*))

(defun ShowLink (theLink)
  "Outputs diagnostic info. for a given link object.
  See FindLink, ShowLinks."

   (if (eq (BPLink-fromNode theLink) *biasNode*)
       (format *stdio*
           "Link <~d>:  [Bias   >~d], weight, delt, prevDelt = ~f ~f ~f~%"
           (BPLink-name theLink)
           (BPNode-name (BPLink-toNode theLink))
           (BPLink-weight theLink)
           (BPLink-delta theLink)
           (BPLink-prevDelta theLink))

   (format *stdio*
```

```
            "Link <~d>:  [~d  >~d], weight, delt, prevDelt = ~f ~f ~f~%"
            (BPLink name theLink)
            (BPNode name (BPLink fromNode theLink))
            (BPNode name (BPLink toNode theLink))
            (BPLink weight theLink)
            (BPLink delta theLink)
            (BPLink prevDelta theLink))))

(defun ShowNode (theNode &optional (verbose p nil))
 "Outputs diagnostic info. for a given node object.  If given a second
 (optional) non nil parameter, gives a long description.
 See ShowNodes, ShowGroupsNodes."

  (format *stdio*
     "Node [~d]: input, output = ~f, ~f~%      ErrorSig, IncomingErrSig = ~f, ~f~%"
     (BPNode name theNode)
     (BPNode input theNode)
     (BPNode output theNode)
     (BPNode errSig theNode)
     (BPNode incomingErrSig theNode))

  (cond
     (verbose p
        (format *stdio* "          InLinks: ")
        (dolist (x (BPNode inLinks theNode))
              (if (BPLink fromNode x)
                 (format *stdio* "<~d>[~d] "
                       (BPLink name x) (BPNode name (BPLink fromNode x)))
                 (format *stdio* "<NIL> ")))
        (terpri *stdio*)

       (format *stdio* "          OutLinks: ")
       (dolist (x (BPNode outLinks theNode))
             (if (BPLink toNode x)
                (format *stdio* "<~d>[~d] "
                       (BPLink name x) (BPNode name (BPLink toNode x)))
                (format *stdio* "<NIL> ")))

       (terpri *stdio*)))
)

(defun ShowGroupsNodes (theGroup &optional (verbose p nil))
 "Prints diagnostic info. for the given list of nodes.  Given a second
 (optional) non nil parameter, gives a long description.
 See ShowNode, ShowNodes."

  (dolist (x theGroup)
        (ShowNode x verbose p)))

(defun ShowNodes (&optional (verbose p nil))
 "Prints diagnostic info. for all the nodes in the net, except the bias node.
 The output is separated into separate sections for each grouping of nodes in
 the net.  If an optional second parameter is given with non nil value, a long
 description is given.
 See ShowGroupsNodes, ShowNode."

  (do ((level list *levels* (cdr level list))
       (cur level)
       (level counter 0 (+ 1 level counter)))
```

```
    ((not level list) t)

    (setq cur_level (car level-list))
    (format *stdio* "~%=====> LEVEL ~d <=====~%" level counter)
    (ShowGroupsNodes cur-level verbose p)))

(defun SetUp424
    (&aux in hid out)                    ; Tmp variables holding the nodes of the
                                         ; input, output, & hidden layers of the net.

"Creates a 4-2-4 network, intended for learning a simple identify encoding:
0001 ->0001, 0010->0010, 0100->0100, 1000->1000.
These input and output patterns are stored in '*inPatts*' and '*outPatts*'."

  (setq *inPatts* '((0.0 0.0 0.0 1.0) (0.0 0.0 1.0 0.0) (0.0 1.0 0.0 0.0) (1.0 0.0 0.0 0.0)))
  (setq *outPatts* *inPatts*)

  (ClearNet)
  (setq in (DefGroup 4 t))
  (setq hid (DefGroup 2))
  (setq out (DefGroup 4))
  (InterConnect in hid)
  (InterConnect hid out)

  (Noise .2)
  (setq *levels* (list in hid out))
  (quote '424Net))

(defun SetUpXOR221
  (&aux in hid out)                      ; Tmp variables holding the nodes of the
                                         ; input, output, & hidden layers of the net.

"Creates a 2-2-1 network, intended for learning the XOR encoding:
00 ->0, 11 ->0, 01 ->1, and 10 ->1.
These input and output patterns are stored in '*inPatts*' and '*outPatts*'."

  (setq *inPatts* '((0.0 0.0) (1.0 1.0) (0.0 1.0) (1.0 0.0)))
  (setq *outPatts* '((0.0) (0.0) (1.0) (1.0)))

  (ClearNet)
  (setq in (DefGroup 2 t))
  (setq hid (DefGroup 2))
  (setq out (DefGroup 1))
  (InterConnect in hid)
  (InterConnect hid out)

  (Noise .2)
  (setq *levels* (list in hid out))
  (quote 'XOR_2_2_1_Net))

(defun SetUpXOR211
  (&aux in hid out)                      ; Tmp variables holding the nodes of the
                                         ; input, output, & hidden layers of the net.

"Creates a 2_1_1 network, intended for learning the XOR encoding:
00 ->0, 11 ->0, 01 ->1, and 10 ->1.
These input and output patterns are stored in '*inPatts*' and '*outPatts*'.
The input layer is connected to both the hidden and output layers."
```

```
  (setq *inPatts* '((0.0 0.0) (1.0 1.0) (0.0 1.0) (1.0 0.0)))
  (setq *outPatts* '((0.0) (0.0) (1.0) (1.0)))

  (ClearNet)
  (setq in (DefGroup 2 t))
  (setq hid (DefGroup 1))
  (setq out (DefGroup 1))
  (InterConnect in hid)
  (InterConnect in out)
  (InterConnect hid out)

  (Noise .2)
  (setq *levels* (list in hid out))
  (quote 'XOR-2-1-1 Net))

(defun FlushOutput (stream)
  (if (typep stream 'stream)
      (force-output stream)
      (force-output)))

(defun doc (varName &optional (varType 'function)
        &aux result)
 "Abbreviated version of 'documentation', returns the doc field associated
 with the functions of BackProp. Can also be used to access the doc field
 of defvar'd variables and defstruct'd data types, but for these the user
 must give a second parameter of either 'variable or 'structure."

 (cond
    ((setq result (documentation varName varType)) result)
    ((setq result (documentation varName 'function)) result)
    ((setq result (documentation varName 'structure)) result)
    ((setq result (documentation varName 'variable)) result)
    (t (setq result nil) nil))
 result)

(defun ShowBehavior
   (
    &key
    (acceptable-diff gBPAcceptableDiff)          ; See Learn f/descript. of this parm.
    (inputs *inPatts*)          ; The net will be run on each of these input
                                ; patterns.
    (outputs *outPatts*)
    (verbose-p t)               ; Flag      output should be verbose.
    &aux
    totError                    ; Sum of squares of error sigs of output
                                ; layer.
    outputLayer)                ; The last level of the net.

 "ShowBehavior <acceptable diff> :inputs <*inPatts*> :outputs <*outPatts*>
        :verbose p <nil>.
 Outputs a table detailing the responses of the net to each of the given
 input patterns. The net's output is compared to the given patterns and the
 results tabulated. If verbose flag is turned on, the listing will be more
 detailed."

 (setq outputLayer (car (last *levels*)))
 (setq totError 0.0)      ; Clear error accumulator.

 (format *stdio* "~% Response of net at epoch ~d :~%" gBPIterCntr)
```

```
(PrintALine *stdio*)

;
; Now we print the results for each input pattern.
;

(do
    ((remainingInPatts inputs (cdr remainingInPatts))
     (remainingOutPatts outputs (cdr remainingOutPatts))
     (curError 0.0)
     (curInPatt nil)
     (curOutPatt nil))
    ((not (and remainingInPatts remainingOutPatts)) t)

   (setq curInPatt (car remainingInPatts))
   (setq curOutPatt (car remainingOutPatts))
   (ForwardPass curInPatt)     ; Run on an input.
   (setq curError              ; Calc response correctness (& set
                               ; incomingSigErr fields).
         (OutputResponse outputLayer curOutPatt))

   (incf totError curError)    ; Accumulate error signals.

   (PrintLevelStats curInPatt "IN:          " "~E " #'identity)
   (PrintLevelStats curOutPatt  "OUT (desired): " "~E " #'identity)
   (PrintLevelStats outputLayer "OUT (actual):  " "~E " #'BPNode-output)
   (PrintLevelStats outputLayer "Difference:    " "~E " #'BPNode-incomingErrSig)
   (format *stdio* "Error (summed squares): ~E~%" curError)
   (if (AcceptablePerformance outputLayer acceptable-diff)
      (write-line "The response to this input was acceptable!" *stdio*))

   (PrintALine *stdio*)
) ; DO (for each input pattern).

(format *stdio* "Total error, for all inputs: ~E~%~%" totError))



(defun PrintALine (outputStream)
    (format outputStream "                                                    ~%"))

(defun PrintLevelStats
   (nodes                      ; Prints stats on this group of nodes.
    prefixString          ; Prefixing 1st line of stats w/this str.
    formatString               ; #format string for printing results of
                               ;   statFunction .
    statFunction               ; Apply this function to each node to get

                          ; the statistic to be printed.
   &aux
   prefixSize)                 ; # of chars in the prefix.

  (setq prefixSize (length prefixString))
  (format *stdio* "~A" prefixString)

  (do ((remaining nodes nodes (cdr remaining nodes))
       (curNode)
       (curOutput)
       (curCursorPos prefixSize))
```

```
      ((not remaining nodes))

      (setq curNode (car remaining nodes))

      (when (> curCursorPos 72)
            (terpri *stdio*)
            (write-string "                    "
                          *stdio* :start 1 :end prefixSize)
            (setq curCursorPos prefixSize))

      (setq curOutput
            (format nil formatString
                    (funcall statFunction curNode)))

      (incf curCursorPos (length curOutput))
      (write-string curOutput *stdio*))
   (terpri *stdio*)

)

(defun OutputResponse
    (outputLayer                    ; List of nodes in output layer.
     desiredOutput                  ; Desired output of given nodes.
     &aux
     error)                         ; Sum of squares of err sigs of each node.

  "Calculates the correctness of a layer's (usually the output layer)
   response.  The activation values of the nodes of the given layer are
   compared against the values in <desiredOutput>, and the differences are
   placed in the incomingErrSig field of each node.
   SEE: BackwardPass, ShowBehavior."

  (setq error 0.0)                  ; Clear error accumulator.

  (do ((output-nodes outputLayer (cdr output-nodes))
       (output node)                ; Output node we're currently examining.
       (outputs-remaining desiredOutput (cdr outputs-remaining))
       (output)                     ; Expected (desired) activ val f/cur node.
       (curErrSig))                 ; Err sig of current node.
      ((not (and output-nodes outputs-remaining))
       nil)

     (setq output-node (car output-nodes))
     (setq output (car outputs-remaining))

; Set the incoming error signal field of each output node to the difference
; between the desired and actual output values.  Actually, the square of it.

     (setq curErrSig
           (setf (BPNode-incomingErrSig output-node)
                 (- (BPNode-output output-node) output)))
     (incf error (* curErrSig curErrSig)))          ; Accum error of all nodes.

  error                         ; RETURN sum of squares of error sigs.
)

(defun SetBreakCount
    (&optional
     (numIters gBPBreakCount))                      ; Will break every      epochs.
```

```
"Sets the 'break' counter to the given value.  BackProp will enter a break
loop every <> epochs. See CheckEpochCounts.  If no parameter is given, the
counter is reset to its default value.  If parameter is <= 0, there'll be no
breaks."

(when (> numIters 0)
      (format *stdio* "Will break every ~d epochs.~%" numIters)
      (FlushOutput *stdio*))

(setq gBPBreakCount (setq gBPBreakCntr numIters)))


(defun SetOutputCount
   (&optional
    (numIters gBPOutputCntr))                    ; Will break every ·  ·· epochs.

"Sets the 'output' counter to the given value.  BackProp will print output
a brief diagnostic message every <> epochs.  If the given value is <= 0, no
such messages will be printed.  If no parameter is given, the
output counter is reset."

(when (> numIters 0)
      (format *stdio*
              "Will print brief diagnostics every ~d epochs.~%" numIters)
      (FlushOutput *stdio*))
(setq gBPOutputCount (setq gBPOutputCntr numIters)))

(defun SetMaxItersCount
   (&optional
    (numIters gBPMaxIterCount))                  ; Will break after ·   · epochs.

"Will break with a warning message after <> epochs; intended as a detector
for 'infinite loops', i.e., nets that won't stabilize.  Defaults to value
already stored in gBPMaxIterCount.  If given value <= 0, there will be no
limit to number of epochs.
See: CheckEpochCounts."

(cond ((> numIters 0)
          (format *stdio*
           "Will break at epoch ~d to avoid non stabilizing nets.~%" numIters)
          (FlushOutput *stdio*))
      (t
        (format *stdio* "WARNING: Running with no epoch limitation.~%")))

(setq gBPMaxIterCount numIters))

(defun ResetEpochCount
   ()

"Resets the epoch counting variable.  This DOES NOT affect the Output and
Break counters.
See: CheckEpochCounts."

(format *stdio* "Resetting epoch count to 0.~%")
(FlushOutput *stdio*)
(setq gBPIterCntr 0))

(defun CheckEpochCounts
```

```
  (curError)                    ; Total error accumulated during last epoch.

"Checks the Break, Output, and infinite loop detector counters. If the
break counter has decremented to 0, BackProp enters a break loop, allowing
the user to examine the program's variables. If the output counter has
decremented to 0, BackProp outputs a brief diagnostic message. If the total
number of epochs exceeds the MaxIterCount, BackProp enters a break loop with
a message indicating that the net may not be stabilizing.
SIDE EFFECT: All these
counters are decremented (or incremented, depending) by this function. This
function MUST be called after each epoch.
SEE: SetOutputCount, SetBreakCount, SetMaxItersCount, ResetEpochCount.
GLOBALS: gBPBreakCount, gBPOutputCount, gBPMaxIterCount, gBPIterCntr,
          gBPOutputCntr, gBPBreakCntr."

  (incf gBPIterCntr)            ; +1
  (decf gBPBreakCntr)
  (decf gBPOutputCntr)

  (when (= gBPOutputCntr 0)
        (setq gBPOutputCntr gBPOutputCount); Reset counter
        (cond
           ((= 0 gBPVerbosity)
            (format *stdio* "Epoch: ~d  Error: ~f~%" gBPIterCntr curError)
                                        ; Output diagnostics.
            (FlushOutput *stdio*))
           ((= 1 gBPVerbosity)
            (ShowBehavior))
           )
  )

  (when (= gBPBreakCntr 0)
        (setq gBPBreakCntr gBPBreakCount) ; Reset counter
        (format *stdio* "Entering a break loop on epoch ~d, as per orders....~%"
                gBPIterCntr)
        (FlushOutput *stdio*)
        (break))

  (when (= gBPIterCntr gBPMaxIterCount)
        (format *stdio* "BREAKing because of too many epochs (~d)....~%"
                gBPIterCntr)
        (break)
        (format *stdio* "Hey, it's your CPU!~%")))


(defun SetVerbosity
  (newVerbosityLevel)           ;The new verbosity level.  0      brief.  1      long.
  "Sets the verbosity of most of the diagnostics.  0==>brief, 1==>long."

  (cond
      ((= newVerbosityLevel 0)
          (format *stdio* "Diagnostics will be brief. (SetVerbosity).~%")
          (setq gBPVerbosity newVerbosityLevel))
      ((= newVerbosityLevel 1)
          (format *stdio* "Diagnostics will be verbose. (SetVerbosity).~%")
          (setq gBPVerbosity newVerbosityLevel))
      (t
         (format *stdio* "SetVerbosity [0 or 1].~%"))))
```

Appendix B: Example Session

```
>(load "backprop")
Loading backprop.o
Finished loading backprop.o
28656

>(SetUp424)
created link <1> from [1] to [7]
created link <2> from [1] to [6]
created link <3> from [1] to [11]
created link <4> from [1] to [10]
created link <5> from [1] to [9]
created link <6> from [1] to [8]
created link <7> from [5] to [7]
created link <8> from [5] to [6]
created link <9> from [4] to [7]
created link <10> from [4] to [6]
created link <11> from [3] to [7]
created link <12> from [3] to [6]
created link <13> from [2] to [7]
created link <14> from [2] to [6]
created link <15> from [7] to [11]
created link <16> from [7] to [10]
created link <17> from [7] to [9]
created link <18> from [7] to [8]
created link <19> from [6] to [11]
created link <20> from [6] to [10]
created link <21> from [6] to [9]
created link <22> from [6] to [8]
'424NET

>(Learn)
Diagnostics will be brief. (SetVerbosity).
Will print brief diagnostics every 10 epochs.
Will break at epoch 10000 to avoid non stabilizing nets.
Resetting epoch count to 0.
Epoch: 10  Error: 3.2069
Epoch: 20  Error: 3.0399
Epoch: 30  Error: 2.9335
Epoch: 40  Error: 2.5886
Epoch: 50  Error: 1.8478
Epoch: 60  Error: 1.3047
Epoch: 70  Error: 1.0190
Epoch: 80  Error: 1.0000
Epoch: 90  Error: 0.9782
Epoch: 100  Error: 0.9296
Epoch: 110  Error: 0.8595
Epoch: 120  Error: 0.7012
Epoch: 130  Error: 0.5147
Epoch: 140  Error: 0.3697
Epoch: 150  Error: 0.1473
Acceptable net successfully created!  Epoch: 154

>(ShowBehavior)

 Response of net at epoch 154 :
```

```
IN:          0.0000 0.0000 0.0000 1.0000
OUT (desired): 0.0000 0.0000 0.0000 1.0000
OUT (actual):  0.1854 0.0802 0.0010 0.8787
Difference:    0.1854 0.0802 0.0010  0.1213
Error (summed squares): 0.0555
The response to this input was acceptable!

IN:          0.0000 0.0000 1.0000 0.0000
OUT (desired): 0.0000 0.0000 1.0000 0.0000
OUT (actual):  0.1625 0.1708 0.8053 0.0010
Difference:    0.1625 0.1708  0.1947 0.0010
Error (summed squares): 0.0935
The response to this input was acceptable!

IN:          0.0000 1.0000 0.0000 0.0000
OUT (desired): 0.0000 1.0000 0.0000 0.0000
OUT (actual):  0.0050 0.8292 0.0210 0.0584
Difference:    0.0050  0.1708 0.0210 0.0584
Error (summed squares): 0.0331
The response to this input was acceptable!

IN:          1.0000 0.0000 0.0000 0.0000
OUT (desired): 1.0000 0.0000 0.0000 0.0000
OUT (actual):  0.8146 0.0072 0.1598 0.0832
Difference:   -0.1854 0.0072 0.1598 0.0832
Error (summed squares): 0.0669
The response to this input was acceptable!

Total error, for all inputs: 0.2489


>(ShowLinks)
Current contents of *links*:
Link <22>: [6   >8], weight, delt, prevDelt = 5.3143 0.0000  0.0217
Link <21>: [6   >9], weight, delt, prevDelt =  6.2999 0.0000 0.0131
Link <20>: [6   >10], weight, delt, prevDelt = 2.8547 0.0000  -0.0080
Link <19>: [6   >11], weight, delt, prevDelt =  -3.4233 0.0000 0.0510
Link <18>: [7   >8], weight, delt, prevDelt =  -4.6973 0.0000 0.0020
Link <17>: [7   >9], weight, delt, prevDelt = 2.8009 0.0000  0.0302
Link <16>: [7   >10], weight, delt, prevDelt = 4.4978 0.0000 0.0040
Link <15>: [7   >11], weight, delt, prevDelt =  -4.2838 0.0000 0.0277
Link <14>: [2   >6], weight, delt, prevDelt = 3.8240 0.0000  0.0068
Link <13>: [2   >7], weight, delt, prevDelt =  -2.8634 0.0000  -0.0039
Link <12>: [3   >6], weight, delt, prevDelt =  4.2436 0.0000 0.0042
Link <11>: [3   >7], weight, delt, prevDelt = 1.9860 0.0000  0.0148
Link <10>: [4   >6], weight, delt, prevDelt = 2.3509 0.0000  0.0031
Link <9>: [4   >7], weight, delt, prevDelt = 4.1783 0.0000  0.0021
Link <8>: [5   >6], weight, delt, prevDelt =  1.8103 0.0000 0.0095
Link <7>: [5   >7], weight, delt, prevDelt =  3.5220 0.0000 0.0153
Link <6>: [Bias   >8], weight, delt, prevDelt =  3.0178 0.0000  0.0028
Link <5>: [Bias   >9], weight, delt, prevDelt =  0.8548 0.0000 0.0210
Link <4>: [Bias   >10], weight, delt, prevDelt =  5.4162 0.0000 0.0196
Link <3>: [Bias   >11], weight, delt, prevDelt = 2.0358 0.0000  0.0034
Link <2>: [Bias   >6], weight, delt, prevDelt =  0.0296 0.0000 0.0038
Link <1>: [Bias   >7], weight, delt, prevDelt =  0.2942 0.0000  0.0054

NIL
```

```
>(ShowNodes)

=====> LEVEL 0 <=====
Node [5]: input, output = 0.0000, 1.0000
          ErrorSig, IncomingErrSig = 0.0000, 0.0000
Node [4]: input, output = 0.0000, 0.0000
          ErrorSig, IncomingErrSig = 0.0000, 0.0000
Node [3]: input, output = 0.0000, 0.0000
          ErrorSig, IncomingErrSig = 0.0000, 0.0000
Node [2]: input, output = 0.0000, 0.0000
          ErrorSig, IncomingErrSig = 0.0000, 0.0000

=====> LEVEL 1 <=====
Node [7]: input, output =  3.8162, 0.0215
          ErrorSig, IncomingErrSig =  0.0017, 0.0000
Node [6]: input, output =  1.8399, 0.1371
          ErrorSig, IncomingErrSig =  0.0035, 0.0000

=====> LEVEL 2 <=====
Node [11]: input, output = 1.4747, 0.8146
           ErrorSig, IncomingErrSig = 0.0323,  0.1854
Node [10]: input, output =  4.9284, 0.0072
           ErrorSig, IncomingErrSig = 0.0061, 0.0072
Node [9]: input, output =  1.6581, 0.1598
          ErrorSig, IncomingErrSig = 0.0000, 0.1598
Node [8]: input, output =  2.3903, 0.0832
          ErrorSig, IncomingErrSig =  0.0138, 0.0832
T

>(PurifyLinks)
NIL

>(Noise .2)
NIL

>(setq *inPatts* '((1 1 1 1) (1 1 0 0) (0 0 1 1 ) (0 0 0 0)))
((1 1 1 1) (1 1 0 0) (0 0 1 1) (0 0 0 0))

>(setq *outPatts* '((1 0 0 0) (0 1 0 0) (0 0 1 0) (0 0 0 1)))
((1 0 0 0) (0 1 0 0) (0 0 1 0) (0 0 0 1))

>(LetSetOutputCount 20)
Will print brief diagnostics every 20 epochs.
20

>(SetBreakCount 50)
Will break every 50 epochs.
50

>(Learn)
Diagnostics will be brief. (SetVerbosity).
Will break every 50 epochs.
Will print brief diagnostics every 20 epochs.
Will break at epoch 10000 to avoid non stabilizing nets.
Resetting epoch count to 0.
Epoch: 20  Error: 3.0557
Epoch: 40  Error: 2.8431
Entering a break loop on epoch 50, as per orders....
```

```
Break.

Broken at EVAL.  Type :H for Help.
>>(ShowBehavior)

 Response of net at epoch 50 :

IN:          1.0000 1.0000 1.0000 1.0000
OUT (desired): 1.0000 0.0000 0.0000 0.0000
OUT (actual):  0.3521 0.3143 0.2953 0.2387
Difference:      0.6479 0.3143 0.2953 0.2387
Error (summed squares): 0.6628

IN:          1.0000 1.0000 0.0000 0.0000
OUT (desired): 0.0000 1.0000 0.0000 0.0000
OUT (actual):  0.3165 0.3682 0.2387 0.2769
Difference:    0.3165   0.6318 0.2387 0.2769
Error (summed squares): 0.6330

IN:          0.0000 0.0000 1.0000 1.0000
OUT (desired): 0.0000 0.0000 1.0000 0.0000
OUT (actual):  0.2210 0.1571 0.4354 0.2973
Difference:    0.2210 0.1571   0.5646 0.2973
Error (summed squares): 0.4807

IN:          0.0000 0.0000 0.0000 0.0000
OUT (desired): 0.0000 0.0000 0.0000 1.0000
OUT (actual):  0.1795 0.2423 0.2911 0.3775
Difference:    0.1795 0.2423 0.2911   0.6225
Error (summed squares): 0.5631

Total error, for all inputs: 2.3396


>>:r
Epoch: 60  Error: 1.5805
Epoch: 80  Error: 0.6113
Epoch: 100  Error: 0.1746
Entering a break loop on epoch 100, as per orders....

Break.

Broken at EVAL.  Type :H for Help.
>>:r
Acceptable net successfully created!  Epoch: 104

>(ShowBehavior)

 Response of net at epoch 104 :

IN:          1.0000 1.0000 1.0000 1.0000
OUT (desired): 1.0000 0.0000 0.0000 0.0000
OUT (actual):  0.8115 0.1519 0.0998 0.0110
Difference:      0.1885 0.1519 0.0998 0.0110
Error (summed squares): 0.0687
The response to this input was acceptable!

IN:          1.0000 1.0000 0.0000 0.0000
OUT (desired): 0.0000 1.0000 0.0000 0.0000
```

```
OUT (actual):  0.1256 0.8264 0.0033 0.1765
Difference:    0.1256  0.1736 0.0033 0.1765
Error (summed squares): 0.0771
The response to this input was acceptable!

IN:          0.0000 0.0000 1.0000 1.0000
OUT (desired): 0.0000 0.0000 1.0000 0.0000
OUT (actual):  0.1947 0.0025 0.8320 0.0962
Difference:    0.1947 0.0025  0.1680 0.0962
Error (summed squares): 0.0754
The response to this input was acceptable!

IN:          0.0000 0.0000 0.0000 0.0000
OUT (desired): 0.0000 0.0000 0.0000 1.0000
OUT (actual):  0.0037 0.0731 0.1016 0.8053
Difference:    0.0037 0.0731 0.1016  0.1947
Error (summed squares): 0.0536
The response to this input was acceptable!

Total error, for all inputs: 0.2747


>(ShowLinks)
Current contents of *links*:
Link <22>:  [6   >8], weight, delt, prevDelt = 2.2017 0.0000  0.0198
Link <21>:  [6   >9], weight, delt, prevDelt = 5.3465 0.0000  0.0297
Link <20>:  [6   >10], weight, delt, prevDelt =  5.9742 0.0000 0.0289
Link <19>:  [6   >11], weight, delt, prevDelt =  2.8851 0.0000 0.0281
Link <18>:  [7   >8], weight, delt, prevDelt = 4.7861 0.0000  0.0016
Link <17>:  [7   >9], weight, delt, prevDelt =  3.2883 0.0000 0.0092
Link <16>:  [7   >10], weight, delt, prevDelt = 2.8566 0.0000  0.0282
Link <15>:  [7   >11], weight, delt, prevDelt =  5.5822 0.0000 0.0321
Link <14>:  [2   >6], weight, delt, prevDelt = 1.9239 0.0000 0.0014
Link <13>:  [2   >7], weight, delt, prevDelt =  2.6946 0.0000 0.0009
Link <12>:  [3   >6], weight, delt, prevDelt = 1.6558 0.0000 0.0014
Link <11>:  [3   >7], weight, delt, prevDelt =  2.8630 0.0000 0.0009
Link <10>:  [4   >6], weight, delt, prevDelt =  3.2085 0.0000 0.0086
Link <9>:  [4   >7], weight, delt, prevDelt =  1.3704 0.0000 0.0021
Link <8>:  [5   >6], weight, delt, prevDelt =  3.1350 0.0000 0.0086
Link <7>:  [5   >7], weight, delt, prevDelt =  1.2590 0.0000 0.0021
Link <6>:  [Bias   >8], weight, delt, prevDelt =  5.0176 0.0000 0.0366
Link <5>:  [Bias   >9], weight, delt, prevDelt =  3.3115 0.0000  0.0027
Link <4>:  [Bias   >10], weight, delt, prevDelt =  0.4671 0.0000 0.0199
Link <3>:  [Bias   >11], weight, delt, prevDelt = 2.1295 0.0000  0.0158
Link <2>:  [Bias   >6], weight, delt, prevDelt = 1.4657 0.0000  0.0120
Link <1>:  [Bias   >7], weight, delt, prevDelt = 3.5992 0.0000  0.0156

NIL
```