The Neural Networks Toolkit

version 2.0

September 1993

1

p1©

Southern Scientific

Introduction

What it is

The Southern Scientific Neural Networks Toolkit is for programmers. It is intended to form the basis for NN designs ranging from the conventional to the bizarre. To do this, the Toolkit was designed using OOP techniques, keeping absolute flexibility in mind, containing the basic objects and methods which are likely to occur in any architecture, but still providing the freedom to try out new insights and ideas with a minimum of programming. It is assumed that the user of this product understands OOP techniques as implemented in Borland Pascal, is familiar with basic linear algebra, can produce derivatives of simple functions and has read basic literature on Backpropagation Neural Networks.

The Toolkit is independent of Windows and DOS, that is, you can use it in both environments.

What it contains

The Toolkit comes in 4 main parts, implemented as Turbo Pascal units, and includes some utilities :

◆**The basic unit**, **NNunit,** containing the 'grandfather' Neuron and Neuralnet objects, from which the user produces descendants suitable for a specific design. Source code is available separately.

◆**The dynamic matrices and vectors unit**, **Dyna2,** which contains objects to deal with the connections between neurons, and provides, as an added bonus, tools for numerical analysis applications which require dynamic matrix data structures. Source code is available separately.

◆**The Backpropagation unit, BPNet2**, implements a standard backpropagation network, and provides an example of the use of the objects in the first two units. Source is provided.

◆**The Brainmaker unit, Brain,** provides access to the structure and weights of a network trained with Brainmaker, a popular product from California Scientific (no relation). Source is provided.

◆**A working windows Backprop application**, SLUG3.EXE, with source, to illustrate the use of the Toolkit. Use SLUGHLP2.HLP for information on how to use SLUG3. You should also look at DOSDEMO2.PAS for a quick look at how easy it is to make a backprop net work with the Toolkit.

◆**Several other small utilities.** Check your distribution disk

An Overview_____

It is useful to think of a neural network in the most general terms, without preconceived notions of connectivity or function. Very many neural network paradigms exist, and to use the Toolkit effectively, you should think of a generic neural network simply as a bag of neurons, each connected to every other neuron, and use this as a foundation for the construction of specific architectures.

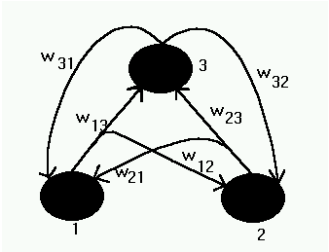
The Neuralnet object This is how the Neuralnet object (the 'grandfather' object) is defined. It provides a starting point when you build a new architecture. The Neuralnet object is a descendant of TCollection, an extremely useful object provided by Borland. Not surprisingly, Neuralnet is a TCollection of Neuron objects (more about these later). The toolkit supplies tools to connect and disconnect neurons, and to group sets of neurons conceptually into what are called ***neuron fields***. Think of fields as subsets of neurons in the neural network.

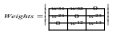
Fields and the fieldlist Fields are pointers to TCollections of neurons, like the neuralnet itself, and a data type definition is provided for this. The Neuralnet object contains a data member called ***fieldlist***, a pointer to a TCollection (again!), which contains all the fields defined in the network. Fields need not be disjoint, and indeed, the whole network can be a field in the fieldlist. The Neuralnet object has methods to deal with fields, and you should use these mechanisms to provide structures like input, hidden and output layers in a network. It will be very helpful to understand the TCollection object, and study the interface sections of the units before you start. Later on, we'll walk through the construction of a Backprop network object to illustrate all this.

The Weights matrix

All neurons have connections of some sort, usually specified by a numeric 'weight' which indicates the strength of connections between neurons. For instance, if neuron 1,

with output O_1 is connected to neuron 2, there is a weight w_{12} which scales the total contribution of neuron 1 to neuron 2 : thus, neuron 2 receives a total contribution of $O_1 * w_{12}$ from neuron 1. If you think of any network, like our basic Neuralnet object, then the calculation of 'what goes into' each neuron is very simply provided by multiplication of a weights matrix by a vector containing all outputs (the vector is a *premultiplier* in our implementation). Each entry, A_i , in the resultant vector is then 'what goes into' neuron i. This is sometimes called the activation of the i'th neuron, and is the value upon which the neuron operates to provide its subsequent output. The figure illustrates the relation between the weights and the neuron numbering.





Thus, the Neuralnet object has a data member called ***Weights***, which is a pointer to a dynamic matrix object, which can shrink and grow with the network, should this be necessary. We will discuss the details of this later.

What is important at this point, is to realise that the entries in the weights matrix completely specify the connectivity of the network, and has N^2 entries for a network with N neurons. For sparsely connected networks, like backprop nets, this is not efficient, but it certainly is necessary in the 'grandfather' Neuralnet object, since we have to cater for all possible constructions. There has to be a number (even if it is zero) for the connection from any neuron to every other neuron.

Fields for input and output Every neural net must be presented with data, and produces a result. Therefore, it makes sense to

include input and output fields of neurons as data members in the 'grandfather' object. These are defined as type neuronfield (pointers to Tcollection of neurons).

The Neuralnet object shouldn't 'overspecialise'. In our judgement, the structures discussed above are the only ones which all neural nets have in common, so these are the only ones in the Neuralnet object.

Neuralnet Methods A detailed description of the methods in the neuralnet object follows below. All of them are concerned with the manipulation of fields, connections, neurons and general design concerns, so that it becomes easy to provide descendants with particular architectures. The Neuralnet object is streamable.

The Neuron object Every neuron receives data, processes it in some way, and provides an output, either into other neurons, or to the outside world. Therefore, a 'grandfather' neuron object must implement data and methods for these requirements.

Neuron data members The Neuron object has data fields for the signalfunction it applies to its total input, the derivative of this function, a scalar to scale incoming data, datastuctures to hold its current state and a history of its associated error.

Neuron methods The 'grandfather neuron does only very simple things : it can change its signalfunction, set its scalar, calculate and report its present state, and fire. The fire method presents its output to the outside world.

Working together The Neuron, Neuralnet, Dynamat (for weigths) and Dynavec (dynamic vector, for data manipulation) objects work together to provide the functionality you need to effectively build efficient neural networks. To work creatively with these, and to derive the maximum benefit from the Toolkit, you should understand the TCollection object well.

It is worthwhile to remember that the Tcollection's entries can point to any data structure at all (preferably objects, otherwise you have to override some useful methods in Tcollection). For instance, if you want to have a rectangular array of neurons, you could, if you understood Tcollections, easily modify the Dynamat object to reference these intuitively - you don't have to use numbers in a Dynamat or Dynavec. See the following full description of how these objects were built.

The Dynamic Vectors and Matrices unit.

On your distribution disk you should find the Borland units **DYNA2.TPU, DYNA2.TPP** and **DYNA2.TPW**, for DOS, DOS protected mode and Windows 3.1 respectively. Both of these contain the same functionality. You need not learn two sets of procedure and variable names. The same source was compiled for DOS and Windows.

THE INTERFACE SECTION

```
const
       dynaErrmsgcount = 10;
       DynaErrMsg   : array[1..dynaerrmsgcount] of string[80]
                         =('<<<Index out of range in dynavec.put>>>',
                           '<<<Index out of range in dynavec.get>>>',
                           '<<<Negative or zero index in
dynavec.expandat>>>',
                           '<<<Index out of range in
dynavec.contractat>>>',
                           '<<<Index out of range in
dynamat.deletecol>>>',
                           '<<<Index out of range in
dynamat.deleterow>>>',
                           '<<<Row index out of range in
dynamat.get>>>',
                           '<<<Row index out of range in
dynamat.put>>>',
                           '<<<Index out of range in
dynamat.getrow>>>',
                           '<<<Index out of range in dynamat.getcol>>>'
                          );
var
    DynaError     : integer;    {flags error conditions}
type
     pfloat     = ^float;
{=-----------------------------------}
     float      = object(tobject)        { A floating point object }
{=-----------------------------------}
         num     : double;
         constructor init(a: double);
         function     getnum: double;
         procedure    putnum( a: double);
         constructor load(var s: tstream);
         procedure    store(var s: tstream); virtual;
         end;
const
Rfloat : tstreamrec = (
         objtype   : 11500;
         vmtlink   : ofs(typeof(float)^);
         load       :@float.load;
         store     :@float.store
         );



type
       Pdynavec = ^dynavec;
```

```
{=----------------------------------}
      dynavec   = object(tcollection)   { A simple collection of float}
{=----------------------------------}
      constructor init(alimit,adelta : integer);
      function   get(i: integer): double;
      procedure put(i: integer; num : double);
      procedure expandat   (i : integer);
      procedure contractat(i : integer);
      function norm : double;
      end;

const
Rdynavec : tstreamrec = (
         objtype   : 11501;
         vmtlink   : ofs(typeof(dynavec)^);
         load       :@dynavec.load;
         store     :@dynavec.store
         );
type
     Pdynamat = ^dynamat;
{=----------------------------------}
      dynaMAT = OBJECT(tobject)
{=----------------------------------}
      nrow              : integer;
      ncol              : integer;
      rows              : pcollection;    {of dynavec }
      cols              : pcollection;    {of dynavec }
      constructor init(maxrow, maxcol : integer);
      constructor load(var s: tstream);
      procedure    store( var s: tstream);
      procedure    addrow(i : integer);
      procedure    addcol(j : integer);
      procedure    deleterow(i : integer);
      procedure    deletecol(j : integer);
      procedure    put(i,j   : integer; value : double);
      function     get(i,j   : integer) : double;
      procedure    getrow(i : integer; var pvec : pdynavec);
      procedure    getcol(j : integer; var pvec : pdynavec);
      destructor   done; virtual;
      end;

const
Rdynamat : tstreamrec = (
         objtype   : 11502;
         vmtlink   : ofs(typeof(dynamat)^);
         load       :@dynamat.load;
         store     :@dynamat.store
         );



type
Pcroupier   = ^Croupier;
{---------------------------}
 croupier   = object(tobject)
{---------------------------}
    index            : pdynavec;
    decksize        : integer;
    constructor init(size   : integer);
    procedure newdeck; virtual;
    function deal( deck : pcollection) : pointer; virtual;
    destructor done; virtual;
    end;
{---------------------------}


function   dynadotprod(a,b : dynavec) : double;
procedure printdynaerror;
```

THE DYNAMIC VECTOR OBJECT

This is simply a descendant of the Tcollection object supplied by Borland (i.e. a collection of floating point objects called Float), with some extra methods to do things that one would want to do with vectors which can shrink and grow. Note that indices start at one, not zero, as in the Tcollection object.

Dynavec Data members

constructor dynavec.init(alimit, adelta : integer);

Initialises the vector by calling the Tcollection init method, and sets all entries in the vector to zero by constructing the floats on the heap and inserting them into the collection. This provides an initialised vector, which can immediately be manipulated with the get and put methods.

procedure dynavec.put(i: integer; num : double);

Changes the value of the i'th entry to num. Posts an error in dynaerror if i doesn't make sense.

function dynavec.get(i: integer): double;

Returns the value of the i'th entry if i is OK, else posts an error in dynaerror.

procedure dynavec.expandat(i : integer);

If i positive, expands the vector at position i. Inserts a zero at index i and shifts all items one index up. Expands after the last item if i is too large. If i < 0, reports an error in dynaerror.

.**procedure dynavec.contractat(i : integer);**

Deletes and disposes the entry with index i and shifts items with higher indices down. If i doesn't make sense, does nothing and posts an error in dynaerror.

function dynavec.norm : double; {returns the norm of the object}

Returns the square root of the dotproduct of the vector with itself.

.

THE DYNAMIC MATRIX OBJECT

The dynamat object is a descendant of tobject, and stores its entries as a matrix in two collections of dynavecs, one for the rows, and one for the columns. Since a Dynavec is an array of pointers, entries are, of course, not duplicated.

Dynamat data members

nrow : integer; Count of number of rows in the matrix.

ncol : integer; Count of number of columns in the matrix.

rows : pcollection; {of dynavec }
A pointer to a TCollection which contains the row vectors. The row vectors are, in turn, Dynavecs.

cols : pcollection; {of dynavec }
A pointer to a TCollection which contains the column vectors. The column vectors are also Dynavecs.

Dynamat methods

constructor dynamat.init(maxrow,maxcol : integer);
Initialises the matrix on the heap with maxcol columns and maxrow rows. All entries are zero.

constructor dynamat.load(var s : tstream);
Reads the matrix from the stream s.

procedure dynamat.store(var s : tstream);
Stores the matrix on the stream s by writing nrow, ncol and then storing the rows from row 1 to row nrow.

procedure dynamat.addrow(i : integer);
Adds a row at rowindex i. Rows with index>i shift up. If i<1 the row is added before row 1, and if i>nrow the row is added as the last row. All entries in the new row are zero. The columns are adjusted to reflect the additional row.

procedure dynamat.addcol(j : integer);

Adds a column to the matrix in the same way as described for addrow.

procedure dynamat.deletecol(j : integer);

Deletes column j from the matrix, and disposes its entries. Each row is adjusted to reflect the change. If j doesn't make sense, posts an error in dynaerror.

procedure dynamat.deleterow(i: integer);

Deletes row i from the matrix in the same way as described for deletecol.

.

function dynamat.get(i,j : integer): double;

Returns the value at row i, column j. If the indices don't make sense, posts an error in neuralerror.

procedure dynamat.put(i,j : integer; value : double);

Sets the entry at row i, column j to *value*.

procedure dynamat.getrow(i : integer; var pvec : pdynavec);

Pvec should be nil on entry - this routine simply sets it to point to row i. If i doesn't make sense, posts an error in dynaerror

procedure dynamat.getcol(j : integer; var pvec : pdynavec);

Returns with pvec pointing at column j. If j doesn't make sense, posts an error in dynaerror

destructor dynamat.done;

Disposes the rows and their entries, then deletes all entries in the columns and disposes the columns. Calls tobject.done.

THE CROUPIER OBJECT

This object was constructed in order to present training data to a neural net in random order. Some nets train better this way. The Croupier deals from a dynavec object (the deck), but does not change it.

Index is a dynavec, and decksize an integer.

constructor Croupier.Init(size : integer);

Initialises decksize to *size* (i.e. records the size of the deck it will deal from) and constructs an index to randomly pick from.

procedure Croupier.newdeck;

Calls index^.freeall and reconstructs index. It is the users responsibility to call this method at the correct time during a long 'dealing' session, i.e. whenever the deck is exhausted.

function Croupier.Deal(deck : pcollection) : pointer;

Returns a pointer to a randomly selected entry in *deck*. The used entry in index is disposed. If index is exhausted, returns NIL. *Deck* is a pointer to any non-empty TCollection. *No effort is made to check the compatibilty of index and deck.*

destructor Croupier.Done;

Disposes index and calls tobject.done

UNIT INITIALISATION

```
begin
      randomize;
      dynaerror   := 0;

                        {Stream Registration}
      registertype(rfloat);
      registertype(rcollection);
      registertype(rdynavec);
      registertype(rdynamat);
end.
```

The Basic unit

On your distribution disk you should find the Borland units **NNUNIT.TPU, NNUNIT.TPP** and **NNUNIT.TPW**, for DOS, DOS protected mode and Windows 3.1 respectively. Both of these contain the same functionality. You need not learn two sets of procedure and variable names. The same source was compiled for DOS and Windows. This unit contains the 'grandfather' objects Neuron and Neuralnet.

THE INTERFACE SECTION

```
{ Neural Nets basic unit.
   VERSION 2.0    June 1993
}
UNIT NNUNIT2;
{F+}
{=========================================================================
=}
                                  INTERFACE
{=========================================================================
=}


{$ifdef windows}
uses objects, dyna2, strings,winprocs, wintypes;
{$else}
uses objects, dyna2;
{$endif}
CONST
       smallnumber           = 1.0e-9;
       neuralerrmsgcount   = 13;
       NeuralErrMsg   : array[1..neuralerrmsgcount] of string[80]
                {1}     =('<<- Index zero or negative in addneuron >>>',
                {2}        '<<- Index out of range in getneuron >>>',
                            '<<- Neuron not in net in deleteneuron >>>',
                {4}        '<<- Index out of range in addfield >>>',
                            '<<- Field not in fieldlist in deletefield
>>>',
                {6}        '<<- Field not in fieldlist in killfield
>>>',
                            '<<- Neuron has no inputs above threshold
>>>',
                {8}        '<<- Datalength doesn''t match neuronfield
>>>',
                            '<<- Field not in fieldlist in connect >>>',
               {10}        '<<- Field not in fieldlist in connectbetween
>>>',
                            '<<- Field not in fieldlist in
setfieldsignal >>>',
               {12}        '<<- Could not allocate space - net is empty
>>>',
               {13}        '<<- Could not make field   >>>'
                          );
```

```
VAR
    NeuralError    : integer;   {flags error conditions }

TYPE
      pnum              = ^double;
      pNeuronstate    = ^Neuronstate;

      Neuronstate     = record
          activation   : double;
          output        : double;
      end;

      Neuronfield     = pcollection;   { of neurons }
      psignalfunc     = ^signalfunc;
      signalfunc      = function(a : double): double;
      funcname         = string[20];

{Signalfunctions}
      signaltype      = (linear,
                         arctangent,
                         tanh,
                         halfsine,
                         step,
                         sigmoid,
                         gaussian,
                         one,
                         zero);

{Derivatives of signalfunctions}
      dsignaltype     = (darctangent,
                         dtanh,
                         dhalfsine,
                         dsigmoid,
                         dgaussian
                        );

CONST
      reststate        : neuronstate = (activation:0;output:0);
      signalnames     : array[signaltype] of funcname =
                                  ('linear',
                                   'arctangent',
                                   'tanh',
                                   'halfsine'   ,
                                   'step',
                                   'sigmoid'    ,
                                   'gaussian'   ,
                                   'one',
                                   'zero' );

     dsignalnames    : array[dsignaltype] of funcname =
                                  ('darctangent',
                                   'dtanhfast',
                                   'dhalfsine'   ,
                                   'dsigmoidfast'    ,
                                   'dgaussian'
                                  );
TYPE

      Pneuron= ^Neuron;
{=----------------------------------}
      Neuron = OBJECT(Tobject)
{=----------------------------------}

          sfunctype   : signaltype;
          sfunc        : signalfunc;   { transfer function}
          dsfunc       : signalfunc;   { derivative of transfer
function}
          scalar       : double;        { scalar for activation}
```

```
          state        : Neuronstate; { unfired; for timing purposes}
          output       : double;        { value at output after firing}
          error        : double;        { current error}
          lasterror    : double;        { previous error}
          constructor init(xfer    : signaltype;
                                initial: Neuronstate);
          constructor load(var s: tstream);
          procedure    store(var s: tstream);
          procedure    setsignal(xfer : signaltype);
          procedure    setscale (s     : double);
          procedure    getstate(var s: Neuronstate);
          procedure    calcstate(sigma     : double);
          {sigma = inner prod of weights and network inputs,
normally...}
          procedure    fire;                 {make output available}
          destructor   done; virtual;
      end;

const
      Rneuron : tstreamrec = (
                objtype        : 11400;
                vmtlink        : ofs(typeof(neuron)^);
                load            : @neuron.load;
                store           : @neuron.store
      );
type
      pneuralnet   = ^neuralnet;
{-------------------------------------}
      Neuralnet    = OBJECT(tcollection)    { of Neuron's }
{-------------------------------------}
    {All substructures are referenced through pointers. }

      Weights       : pdynamat;
      fieldlist    : pcollection;   {each entry points to a
                                          collection of neurons -
                                        }
      inputfield : neuronfield; { pointer to input collection }
      outputfield: neuronfield; { pointer to output collection }

      constructor init(total: integer);
      constructor load(var s: tstream);
      procedure    store(var s: tstream);
      procedure    addneuron(i : integer; var aneuron : pneuron);
      procedure    getneuron(i : integer; var aneuron : pneuron);
      procedure    calcallstates; virtual;
      procedure    deleteneuron(var aneuron : pneuron);
      procedure    addfield(var field        : neuronfield;
                            startat, endat : integer);
      procedure    deletefield(var field : neuronfield);
      procedure    fireall;
      procedure    killfield(var field    : neuronfield);
      procedure    getinputsof(thisone    : pneuron;
                               threshold : double;
                               var field : neuronfield); virtual;
      procedure    presentinputto(thefield   : neuronfield;
                                  thedata    : pdynavec);
      procedure    connect(var f:neuronfield; weight: double);
      procedure    disconnect(var f:neuronfield);
      procedure    connectbetween(var from,into: neuronfield;
                                       weight: double);
      procedure    disconnectbetween(var from,into: neuronfield);
      procedure    propagate; virtual;
      procedure    randomweights(alimit : double);
      procedure    nofeedback;
      procedure    setfieldsignal(var field : neuronfield;s :
signaltype);
      destructor   done;   virtual;
    end;
```

```
const
      Rneuralnet : tstreamrec = (
                objtype        : 11401;
                vmtlink        : ofs(typeof(neuralnet)^);
                load            : @neuralnet.load;
                store           : @neuralnet.store
      );
                                                    {prefix 'f' => signal
function
                                                      prefix 'fd' =>
derivative
                                                     }
function flinear(a: double): double;
function farctan(a:double): double;
function fdarctan(a:double): double;
function ftanh(a: double): double;
function fdtanhfast(tanhx: double) : double;
function fhalfsine(a: double): double;
function fdhalfsine(a: double): double;
function fstep(a: double): double;
function fsigmoid(a: double): double;
function fdsigmoidfast(sigx: double): double;
function fgaussian(a: double): double;
function fdgaussian(a: double): double;
function fone(a : double)       : double;    {always one. For offset
neurons}
function fzero(a: double): double;
function findsignalfunc(deriv: boolean; ftype : signaltype): pointer;

procedure printneuralerror;

{   IMPLEMENTATION... }

                                          { Manage heap alloc errors }
{-------------------}
function NeuralHeapError(size : word): integer; far;
{-----------------}
    {Set it up so that allocation errors do not abort,
     but return a nil pointer
    }
begin
      Neuralheaperror := 1;
end;
 {-------------------------- UNIT INITIALIZATION
-----------------------}

begin
      neuralerror := 0;
      randomize;
          {Stream registration}
      registertype(Rneuron);
      registertype(Rneuralnet);
      Heaperror := @NeuralHeapError;
end.
```

THE NEURON OBJECT

The Neuron object is a descendant of TObject. It encapsulates the behaviour of a generic neuron.

Neuron data members

Sfunctype : Signaltype Contains the signal funtion type used by the neuron. Signaltype is an enumerated type with possible values linear, arctangent, halfsine, step, sigmoid and gaussian.

Sfunc : Signalfunc A procedural variable denoting the signalfunction to call. Sfunc takes one parameter of type double (most often the activation of the neuron) and returns a double.

dSfunc : Signalfunc A procedural variable denoting the function which returns the derivative of Sfunc. dSfunc takes one parameter of type double and returns a double.

Scalar : double Contains a scalar for scaling of the activation before transformation by sfunc.

State : neuronstate A record holding the current state (activation and output) of the neuron. *State.output* is the value that appears at *neuron.output* after fire is called. This variable is a 'buffer' for the neuron state, so that it may remain hidden from the network until the algorithm requires a new output, and the neuron formally fires. This is useful for timing purposes in some networks.

Output : double The output from the neuron available to the network for interaction with other neurons. Becomes updated when the neuron fires.

Neuron methods

constructor Neuron.init(xfer : signaltype; initial: neuronstate);

The neuron is initialized by specifying a signalfunction to be *xfer* and the initial neuron state to be *neuronstate* (see the

constant 'reststate'). Calls setsignal, sets output, error and lasterror to zero and sets scalar to 1.0.

procedure Neuron.setsignal(xfer : signaltype);

Sets sfunctype to xfer and calls findsignalfunc to establish the address of the signalfunction. Sets Sfunc and dSfunc to the correct functions. After this call, the neuron uses Sfunc to calculate its output. The user may use dSfunc as necessary, e.g. in a training algorithm.

procedure Neuron.setscale (s : double);

Simply sets scalar to s.

procedure Neuron.getstate(var s: Neuronstate);

Returns the current state of the neuron in s.

procedure Neuron.calcstate(sigma : double);

Sigma = inner prod of weights and inputs to this neuron. Sets activation to sigma and calls the signalfunction set by setsignal with parameter scalar*sigma.

procedure Neuron.fire; Sets neuron.output to the current value in state.output, thereby making the output of the neuron available to the outside world

.

THE NEURALNET OBJECT

The Neuralnet objects is a descendant of TCollection. It contains the neurons in the net, and methods for their manipulation into any specific type of network. This object is probably not very useful as it stands, and is intended to provide basic services for descendants.

NEURALNET FIELDS

Weights : pdynamat;

Pointer to the weights matrix. See the unit DYNA2.

fieldlist: pcollection;

Fieldlist points at a collection of neuronfields(each neuronfield is a pointer to a collection of neurons). These fields represent collections of neurons that the user considers to be logical units, such as an input field, hidden field and output field. Neuralnet methods can access and manipulate fields in this list. Once fields are inserted into fieldlist, the neuralnet object assumes responsibility for their manipulation and disposal. It is wise to use only methods of the neuralnet object to manipulate a field of neurons after it becomes the property of the network. Note that it may sometimes be useful to insert the whole network into fieldlist.

inputfield : neuronfield;
outputfield : neuronfield;

Pointers to output and input fields. These are NIL after the init method is called, and are provided for convenience, since most nets have them. You need not use them.

NEURALNET METHODS

Constructor Neuralnet.init(total: integer);

The number of neurons specified in *total* are created on the heap with the linear signal function and in the reststate. The neurons are inserted into the collection. The weights

matrix is created on the heap with dimensions (*total*,*total*) and each entry is set to 1.0. The fieldlist is created on the heap with space for 3 entries. Inputfield and outputfield are set to NIL, and are NOT inserted in fieldlist. If an error occurs with allocation of space, an error is posted in neuralerror.

constructor neuralnet.load(var s: tstream);

Loads a network from the stream *s*. Calls Tcollection.load, then uses get to laod the weights matrix. Subsequently does the following : Reads the number of neurons, and the indices in neuralnet, of the neurons in each field, reconstructs the fields and updates the fieldlist. Reads indices in fieldlist of the input and output felds and reconstructs these fields.

procedure neuralnet.store(var s: tstream);

Stores the network on a stream. Performs the following sequence : Calls the inherited store method and writes the weights matrix. Writes the number of fields. For each field, writes the number of neurons in the field, and for each of these neurons, its index in neuralnet. Writes the indices in fieldlist of inputfield and outputfield to the stream.

procedure Neuralnet.addneuron(i : integer; var aneuron : pneuron); virtual;

Makes a new neuron, adds it at position i in the net (neurons with numbers >= i move one up) and fixes the weights matrix. On exit, aneuron points to the new, completely disconnected neuron, the i'th in the net (indexof(*aneuron*) = i-1). Disposing of the new neuron is the net's responsibility. Aneuron is NIL on failure. If i doesn't make sense, an error is posted in neuralerror. The new neuron doesn't belong to any of the fields in fieldlist.

procedure Neuralnet.getneuron(i : integer; var aneuron : pneuron); virtual;

Returns with aneuron pointing to neuron # i in the net, i.e neuron with index i-1. Aneuron is NIL if i doesn't make sense.

procedure Neuralnet.deleteneuron(var aneuron : pneuron);virtual;

Deletes and disposes the neuron from the net and deletes it from any fields in fieldlist. Fixes weights matrix. If the neuron is not in the net, nothing is done and an error is posted in neuralerror.

procedure Neuralnet.addfield(var field: neuronfield; startat,endat : integer);virtual;

On entry, *field* points to nothing. A field is initialized, neurons are inserted and the new field inserted into the fieldlist. The new field contains neurons from # startat to # endat (counting from 1) inclusive, in the network. No neurons are created. Disposing the field becomes the responsibility of the network. If startat and endat do not index neurons in the network, nothing is done and an error is posted in neuralerror.

procedure Neuralnet.deletefield(var field : neuronfield); virtual;

Removes a *field* from the fieldlist. The items in *field* are deleted from *field* (not disposed) and the field is disposed of. *Field* is NIL on exit if successfull. If *field* is not in fieldlist, nothing is done, and an error is posted in neuralerror.

procedure Neuralnet.killfield(var field : neuronfield); virtual;

Removes *field* from fieldlist and deletes and disposes neurons in *field* from the net by calling deleteneuron (i.e. weights matrix is corrected, and errors reported). Deletes all items in *field*. Disposes *field* and returns nil in *field* if sucessfull. If *field* is not in fieldlist, nothing is done and an error is flagged in neuralerror.

procedure Neuralnet.getinputsof(thisone : pneuron;
threshold : double; var field : neuronfield); virtual;

Finds neurons with absolute value of connections (weights) to *thisone* greater than threshold. Assumes *field* is nil on entry. Makes a new field, returns all neurons that meet this

criterion in *field*. *Field* is inserted into fieldlist. Interprets 2nd index of weights matrix as destination - weights(i,j) means from neuron i into neuron j. If no neurons meet the criterion, nothing is done, field is NIL on exit and an error is posted in neuralerror. May be slow!

procedure Neuralnet.presentinputto(thefield : neuronfield; thedata : pdynavec); virtual;

Presents numeric data in *thedata* to *thefield*, and calculates the new state of each neuron in *thefield*. Does not fire the neurons. If the number of items in thefield is not the same as the number of items in thedata, nothing is done and an error is posted in neuralerror. See also *neuralnet.propagate.*

procedure Neuralnet.connect(var f:neuronfield; weight: double);virtual;

Fully connects a field of neurons by setting the relevant entries in the weights matrix to weight. If *f* is not in fieldlist, does nothing and posts an error in neuralerror.

procedure Neuralnet.disconnect(var f: neuronfield);virtual;

Fully disconnects a field of neurons. Simply calls connect with a weight parameter of 0.0.

procedure Neuralnet.connectbetween(var from, into: neuronfield; weight: double); virtual;

Completely connects two neuronfields in one direction only by placing *weight* in the relevant positions of the weights matrix. Thus, every neuron in *from* now propagates data to all neurons in *into* . Does not remove existing connections in the other direction. If either neuronfield is not in fieldlist, does nothing and posts an error in neuralerror.

procedure Neuralnet.disconnectbetween(var from,into: neuronfield); virtual;

Calls connectbetween with a weight parameter of 0.0;

procedure Neuralnet.propagate; virtual;

Fires all neurons, then calculates all new states. Calls fireall and calcallstates.

procedure Neuralnet.randomweights(alimit : double); virtual;

Randomizes all entries in the weights matrix to a random value between -limit...+limit with resolution 1/1000 of this interval.

procedure Neuralnet.nofeedback; virtual;

Sets all entries on the diagonal of the weights matrix to 0.0, thus preventing all neurons from feeding directly back into themselves.

procedure Neuralnet.setfieldsignal(var field : neuronfield; s : signaltype); virtual;

Sets the signalfunction for all neurons in field to s. If field is not in fieldlist, does nothing and posts an error in neuralerror.

procedure Neuralnet.fireall ; virtual;

Fires all neurons in the net by calling neuron.fire for each one.

procedure Neuralnet.calcallstates; virtual;

Calculates the new state of each neuron. Calculates dotproducts of outputs and connected weights for each neuron - i.e. for neuron j, calculate Sum(over i) of [output(i)*weights(i,j)] and calculate a new activation for neuron j. For neurons which do not calculate their activation by the dotproduct of its inputs and weights, this method should be overridden.

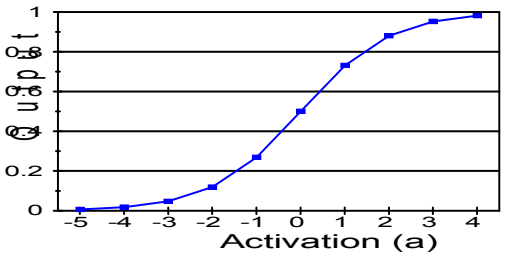
destructor Neuralnet.done; virtual;

Disposes the weights matrix. Empties fieldlist and all fields in fieldlist and disposes these. Calls Tcollection.done.

The Signalfunction

This is the function applied by the neuron to its activation value to produce its output. This function is often called by other names, but we will use **signal function or transfer function** throughout. The most commonly used signalfunction is probably the sigmoid function, shown below. Many others exist, and you can experiment with your own.

The sigmoid



$Sigmoid(a) = \frac{1}{1+\exp(-a)}$

How the signalfunctions work

The Toolkit makes use of procedural variables to handle the assignment of signalfunctions to neurons. When you study the interface section of NNUnit2, you will see that we've implemented a type called signalfunc, as follows:

```
TYPE
      signalfunc      = function(a : double): double;
```

This allows a series of signalfunctions of this type to be written, and we define corresponding enumerated types , as shown below :

```
Type
   signaltype      = (linear,
                      arctangent,
                      tanh,
                      halfsine,
                      step,
                      sigmoid,
                      gaussian,
                      one,
                      zero);
{Derivatives of signalfunctions}
   dsignaltype     = (darctangent,
                      dtanh,
                      dhalfsine,
                      dsigmoid,
                      dgaussian
                     );
```

The setsignal method This means you can now pass the neuron a variable of type signaltype, and ask it to change its transfer function to the indicated type. The Neuron object has a method called ***setsignal*** which does this in conjunction with a function called ***findsignalfunc***.

They look like this :

```
{-------------------------------}
procedure neuron.setsignal(xfer : signaltype);
{-------------------------------}

                   { Changes the neuron's signal function.
                   }
begin
       sfunctype   := xfer;
       @sfunc      := findsignalfunc(false,xfer);
       @dsfunc     := findsignalfunc(true,xfer);
end;

{-------------------------------}
function findsignalfunc(deriv : boolean; ftype : signaltype): pointer;
{-------------------------------}

                                  {If deriv is true, returns pointer
                                   to the derivative function
                                   of ftype
                                   NB - See p 55 of programmers guide
TPW
                                  }
begin
    if not deriv then
       case ftype of
          linear     : findsignalfunc := (@flinear);
          arctangent: findsignalfunc := (@farctan);
          tanh        : findsignalfunc := (@ftanh);
          halfsine   : findsignalfunc := (@fhalfsine);
          step        : findsignalfunc := (@fstep);
          sigmoid    : findsignalfunc := (@fsigmoid);
          gaussian   : findsignalfunc := (@fgaussian);
          one         : findsignalfunc := (@fone);
          zero        : findsignalfunc := (@fzero);
       end
    else
       case ftype of
          linear     : findsignalfunc := (@fone);
          arctangent: findsignalfunc := (@fdarctan);
          tanh        : findsignalfunc := (@fdtanhfast);
          halfsine   : findsignalfunc := (@fdhalfsine);
          step        : findsignalfunc := (@fzero);
          sigmoid    : findsignalfunc := (@fdsigmoidfast);
          gaussian   : findsignalfunc := (@fdgaussian);
          one         : findsignalfunc := (@fzero);
          zero        : findsignalfunc := (@fzero);
       end;
end;
```

This means you can 'ask' the neuron to change its signal function, and it will look up the address of the function and its derivative. After this call, the neuron will use these

functions to calculate its new state (see *Neuron.calcstate*). You can now have your neurons adjust their signalfunctions while your network runs. You can, of course, also mix different signal functions in the same network - the neuron object knows what to do, and you needn't worry about it after your call to *Neuron.setsignal*.

Your own signalfunctions

If you write your own signal function and derivative, there are several ways to implement it. For example, you can simply override *Neuron.setsignal* and replace it with similar code, where the call to *findsignalfunction* is replaced with a call to your own version.

Important !

Some signal functions defined in the Toolkit (the sigmoid and hyperbolic tangent functions), have derivatives defined in terms of the *function value*, not the function argument, making calculation of the derivative faster if the function itself is already known. For example, the derivative, Der(x) of the sigmoid function is defined as :

Der(x) = sigmoid(x){ 1-sigmoid(x) }

so that ***the parameter passed to the derivative is <u>not</u> x, but Sigmoid(x)***. This is useful in situations like backprop training, where the derivative is called for after the neuron fires (i.e. Sigmoid(x) is known). Take care to remember this if you use the provided *findsignalfunc* function.

The signaltypestypes each have a ascii string associated with them, in case you need these in a dialog or list box :

```
CONST
      reststate        : neuronstate = (activation:0;output:0);
      signalnames      : array[signaltype] of funcname =
                                    ('linear',
                                     'arctangent',
                                     'tanh',
                                     'halfsine'   ,
                                     'step',
                                     'sigmoid'    ,
                                     'gaussian'   ,
                                     'one',
                                     'zero' );

     dsignalnames     : array[dsignaltype] of funcname =
                                    ('darctangent',
                                     'dtanhfast',
                                     'dhalfsine'   ,
                                     'dsigmoidfast'    ,
                                     'dgaussian'
```

```
                                        );
```

Provided Signalfunctions

Here are the signalfunctions provided in NNUnit2 :

```
{--------------------------------}
function farctan(a: double): double;
{--------------------------------}
begin
       farctan := 2.0/pi*arctan(a);         { ...limits are -1 and 1 }
end;
{--------------------------------}
function fdarctan(a: double): double;
{--------------------------------}
begin
       fdarctan := 1.0/(1.0+a*a)
end;
{--------------------------------}
function ftanh(a: double): double;
{--------------------------------}
var
   e,inv   : double;
begin
      e     := exp(a);
      inv   := 1.0/e;
      ftanh:= (e-inv)/(e+inv);
end;
{--------------------------------}
function fdtanhfast(tanhx: double): double; {tanhx is ftanh(x) }
{--------------------------------}
begin
      fdtanhfast := (1.0-tanhx*tanhx);
end;
{--------------------------------}
function fsigmoid(a: double): double;
{--------------------------------}
begin
      fsigmoid := 1.0/(1.0 + exp(-a));
end;
{--------------------------------}
function fdsigmoidfast(sigx: double): double;   {sigx is fsigmoid(x) }
{--------------------------------}
begin
      fdsigmoidfast := sigx*(1-sigx);
end;
{--------------------------------}
function fgaussian(a: double): double;
{--------------------------------}
begin
      fgaussian := exp(-a*a);
end;

{--------------------------------}
function fdgaussian(a: double): double;
{--------------------------------}
begin
      fdgaussian:= -2.0*a*fgaussian(a);
end;
{--------------------------------}
function flinear(a:double): double;
{--------------------------------}
begin
            flinear := a;
end;
```

```
{--------------------------------}
function fhalfsine(a: double): double;
{--------------------------------}
begin
       if (a > pi/2.0)
          then fhalfsine := 1.0
       else
          if (a < -pi/2.0)
              then fhalfsine := -1.0
          else
              fhalfsine := sin(a);
end;
{--------------------------------}
function fdhalfsine(a: double): double; {Cheat with derivative}
{--------------------------------}
const
       threshold = pi/2;
begin
       if (a > threshold)
          then fdhalfsine := 1.0
       else
          if (a < -threshold)
              then fdhalfsine := -1.0
          else
              fdhalfsine := cos(a);
end;
{--------------------------------}
function fstep(a: double): double;
{--------------------------------}
begin
       if a<0 then fstep := -1 else fstep := 1;
end;
{--------------------------------}
function fone(a: double): double;
{--------------------------------}
begin
       fone := 1.0;
end;
--------------------------------}
function fzero(a: double): double;
{--------------------------------}
begin
       fzero := 0.0;
end;
```

The BPNet2 Unit

On your distribution disk, you should find Borland units **BPNET2.TPU, BPNET2.TPP** and **BPNET2.TPW**, for DOS, DOS protected mode and Windows 3.1 respectively. You need not learn two sets of procedure and variable names. The same source was compiled for DOS and Windows.

In this unit, a simple backpropagation network object (SimpleBPNet) is implemented. This serves as an illustration of the Toolkit's use. Source is included on your distribution disk.

The Interface section

```
UNIT Bpnet2;
{F+}


      {   IMPLEMENTS SOME BACKPROP NETS. REQUIRES THAT SIGNALFUNCTION
          DERIVATIVE FUNCTION TAKE FUNCTION ITSELF AS PARAMETER. SEE
          UNIT NNunit.
      }

                                          {---------}
                                           INTERFACE
                                          {---------}

{$IFDEF WINDOWS}
uses objects,nnunit2,dyna2;
{$ELSE}
uses objects,nnunit2,dyna2;
{$ENDIF}



TYPE

       Psimplebpnet   = ^simplebpnet;
{-------------------------------------}
      simpleBPnet    = OBJECT(neuralnet)
{-------------------------------------}

                          {A simple 3 layer Backpropnet with momentum}

       learn            : double;
       momen            : double;
       deltaW           : pdynamat;
       offset           : neuronfield;
       hiddenfield    : neuronfield;
       countin,
       counthidden,
       countout       : word;
       constructor init(incount,hiddencount,outcount : word;
                        lcoeff,momentum                  :
double);
```

```
        constructor load(var s : tstream);
        procedure    store(var s : tstream);
        procedure setconnections;
        procedure shake(a : double);
        procedure feedforward(datavec : pdynavec);virtual;
        procedure calcallstates; virtual;
        procedure backprop(var upperfield,lowerfield : neuronfield);
        procedure backpropall(errorvec    : pdynavec);
        procedure getdeltaweights(lcoeff,momentum   : double);
        procedure train(errorvec    : pdynavec);
        destructor done;   virtual;
      end;


const

       RsimpleBPnet : tstreamrec = (
              objtype   : 11402;
              vmtlink   : ofs(typeof(simplebpnet)^);
              load      : @simplebpnet.load;
              store     : @simplebpnet.store
       );

type


        PFastbpnet   = ^Fastbpnet;
{-------------------------------------}
       FastBPnet    = OBJECT(simpleBPnet)
{-------------------------------------}

                             {A fast 3 layer Backpropnet with momentum}

        modfactor        : double;
        constructor init(incount,hiddencount,outcount : integer;
                               lcoeff,momentum,kmod            :
double);
        procedure backpropall(errorvec    : pdynavec);
        procedure getdeltaweights(lcoeff,momentum   : double);
      end;



{                        ---------- Unit initialization ---------- }

begin              {Stream registration etc }

       registertype(Rsimplebpnet);
end.
```

The SimplebpNet object

This object is a descendant of Neuralnet. It overrides some of Neuralnet's methods, and adds others to implement backpropagation of errors, changing of weights and the use of a momentum term. The source code for this object is provided on your distribution diskette. We give a detailed description of this construction to illustrate the use of the base objects.

SimpleBPNet data members

learn : double; The learning rate of the network. This parameter determines the stepsize when adjusting weights.

momen : double;

The momentum term. This gives the net the capability to 'remember' the direction of the step taken during the previous training step, and prevents the optimization achieved in the previous step being completely undone in the current training step.

deltaW : pdynamat;

This is a dynamat which holds the changes to be made to the weights matrix during the current training step. Also used to implement momentum, since it holds the *previous* change to the weights matrix on entry to any training step. This is required for the momentum term.

offset : neuronfield;

Most backprop nets have an offset neuron. This is a neuron with a constant output of 1, with trainable weights to the hidden and output layers.

hiddenfield : neuronfield;

The neuron field holding the hidden layer's neurons. Remember that the parent object has input and output fields.

countin, counthidden, countout : word;

These simply reflect the number of neurons in each layer. The sum of these numbers is one less than the total number of neurons in the network when there is an offset neuron.

SimpleBPnet Methods

constructor SimpleBPNet.init (incount, hiddencount, outcount: word; lcoeff,momentum : double);

Incount , hiddencount and outcount specify the number of input,hidden, and output layer neurons respectively. Lcoeff and momentum are the learning coefficient and the momentum parameter. *Neuralnet.init* is called to initialise the net with one extra neuron (the offset neuron). The other data members are set , and the four neuron fields are prepared by calls to addfield (the fourth field contains only the offset neuron). The first *incount* neurons are in *inputfield*, the next *hiddencount* neurons in *hiddenfield*, the next *outcount* neurons in *outputfield*, and the last neuron in the net is placed in *offsetfield*. The code looks like this :

```
neuralnet.init(incount+hiddencount+outcount+1);
countin        := incount;
counthidden   := hiddencount;
countout       := outcount;
                                          {fully connected...}
                                         {insert fields}

addfield(inputfield,1,incount);
addfield(hiddenfield,incount+1, incount+hiddencount);
addfield(outputfield,incount+hiddencount+1,count-1);
addfield(offset,count,count);
```

After this, the signalfunctions for each field are specified and the structure of the net is provided by a call to the *setconnections* method :

```
setfieldsignal(hiddenfield,sigmoid);
setfieldsignal(outputfield,sigmoid);
setfieldsignal(offset,one);
setconnections;
```

Setconnections makes sure the net feeds from inputfield into hiddenfield into outputfield (i.e. a feedforward net) and removes feedback and connections within fields. Finally,

the output neuron is switched on and the DeltaW matrix is initialized.

constructor SimpleBPNet.load(var s : tstream);

This constructor reads a SimpleBPNet from a stream by calling neuralnet.load. Learn and momen are read directly, followed by the deltaW matrix.

```
{--------------------------------}
constructor simplebpnet.load( var s: tstream);
{--------------------------------}
var
    i,j             : integer;
begin
       neuralnet.load(s);
       s.read(learn,sizeof(learn));
       s.read(momen,sizeof(momen));
       deltaw := pdynamat(s.get);
```

Remember that fieldlist is now available (from neuralnet). All that remains is to decide which fields in fieldlist are the offset and hidden fields (the input and output fields were sorted out when neuralnet was loaded from the stream). The next two items on the stream are these indexes, followed by the neuroncounts in the fields:

```
       s.read(i,sizeof(i));
       s.read(j,sizeof(j));
       if (i> -1) then offset := fieldlist^.at(i);
       if (j> -1) then hiddenfield := fieldlist^.at(j);
       s.read(countin, sizeof(countin));
       s.read(counthidden, sizeof(counthidden));
       s.read(countout, sizeof(countout));

end;
```

procedure SimpleBPNet.store(var s : tstream);

This is simply the previous procedure reversed :

```
{--------------------------------}
procedure simplebpnet.store( var s: tstream);
{--------------------------------}
var
    i,j             : integer;
begin
       neuralnet.store(s);
       s.write(learn,sizeof(learn));
       s.write(momen,sizeof(momen));
       s.put(deltaw);
       i := fieldlist^.indexof(offset);
       j := fieldlist^.indexof(hiddenfield);
       s.write(i,sizeof(i));
       s.write(j,sizeof(j));
       s.write(countin, sizeof(countin));
       s.write(counthidden, sizeof(counthidden));
       s.write(countout, sizeof(countout));
end;
```

procedure SimpleBPNet.setconnections;

This procedure illustrates how easy it has become to make a net do exactly what you want.

```
{--------------------------------}
procedure simpleBPnet.setconnections; {connect feedforward net}
{--------------------------------}
begin
       nofeedback;
       disconnectbetween(inputfield,outputfield);
       disconnectbetween(outputfield,inputfield);
       disconnectbetween(outputfield,hiddenfield);
       disconnectbetween(hiddenfield,inputfield);

       disconnectbetween(offset,inputfield);
       disconnectbetween(inputfield,offset);
       disconnectbetween(hiddenfield,offset);
       disconnectbetween(outputfield,offset);

       disconnect(inputfield);
       disconnect(outputfield);
       disconnect(hiddenfield);

end;
```

procedure SimpleBPNet.shake(a : double);

Perturbs the current weights by between -a and a away from their current values.

```
{--------------------------------}
procedure simpleBPnet.shake(a : double);
{--------------------------------}
const
       resol    = 1000;
var
    i,j     : integer;
    factor : double;
    range   : double;
begin
       range     := 2*a;
       factor    := range/resol;
       for i := 1 to count do
            for j := 1 to count do
            weights^.put(i,j, weights^.get(i,j) +
                                    random(resol)*factor-a);
       setconnections;
end;
```

procedure SimpleBPNet.feedforward(datavec : pdynavec);virtual;

This method allows data to be propagated through the net, from the input layer until the output layer reflects the result of the current input:

```
{--------------------------------}
procedure simpleBPnet.feedforward(datavec : pdynavec);
{--------------------------------}
var
```

```
    j               : integer;
begin

       presentinputto(inputfield,datavec);
       for j := 1 to 3 do
          begin
          propagate;         {call fireall & calcallstates}
          end;
end;
```

procedure SimpleBPNet.calcallstates; virtual;

This procedure 'implements' the weights matrix connectivity by calculating each neuron's activation by inner product of only the relevant weights and outputs.

Here, we've overridden the Neuralnet method, since it's too wasteful - it uses the whole weights matrix.

```
{--------------------------------}
procedure simpleBPnet.calcallstates;
{--------------------------------}
var
    source, dest,temp        : pneuron;
    lowerfield, upperfield : neuronfield;

procedure calc;
var
    i,j,k,l                     : integer;
    sum,a                       : double;
begin
       for j := 1 to (upperfield^.count) do            { Destination }
            begin
            dest := upperfield^.at(j-1);              {get neuron}
            l     := indexof(dest)+1;                   {number in net}
            sum   := 0.0;                                  {initialize
dotproduct}
            for i := 1 to lowerfield^.count do      {calc dotprod}
                begin
                source := lowerfield^.at(i-1);
                k        := indexof(source)+1;
                a        := weights^.get(k,l);
                sum := sum + source^.output * a ;
                end;
            dest^.calcstate(sum); {new state}
            end;
end;

begin
       temp         := offset^.at(0); {get offset neuron}

       lowerfield := inputfield;     {do first layer}
       lowerfield^.insert(temp);     {put offset in lowerfield}
       upperfield := hiddenfield;
       calc;
       lowerfield^.delete(temp);     {remove offset from lower layer}

       lowerfield := hiddenfield;    {second layer}
       lowerfield^.insert(temp);
       upperfield := outputfield;
       calc;
       lowerfield^.delete(temp);
```

```
end;
```

procedure SimpleBPNet.backprop(var upperfield,lowerfield : neuronfield);

This is the key procedure in a backprop net. It propagates errors form an 'upper field' of neurons (where each neuron's error is known) to a 'lower field' of neurons (where each neuron's error is to assigned by this procedure). In this implementation, derivatives of signal functions of lowerfield are assumed to have been defined in terms of the signalfunction itself (see the earlier discussion of signalfunctions).

```
{--------------------------------}
procedure simpleBPnet.backprop(var upperfield,lowerfield :
neuronfield);
{--------------------------------}

                    {Propagates error back from upperfield
                     to lowerfield.   Upperfield errors have
                     been assigned on entry. Signalfunctions
                     MUST be sigmoid or tanh.
                    }

var
    i,j,k,l          : integer;
    field            : neuronfield;
    lower,upper     : pneuron;
    nstate           : neuronstate;
    thiserror        : double;
    sum               : double;

begin
       for i := 1 to lowerfield^.count do
              begin
              sum         := 0.0;
              lower      := lowerfield^.at(i-1);
              thiserror := lower^.dsfunc(lower^.output);
              l           := simplebpnet.indexof(lower) + 1;
              for j := 1 to upperfield^.count do { ...for every upper
neuron}
                   begin
                   upper     := upperfield^.at(j-1);
                   k          := simplebpnet.indexof(upper) + 1;
                   sum        := sum + weights^.get(l,k)*upper^.error;
                   end;

              with lower^ do
                    begin
                    lasterror := error;
                    error      := thiserror*sum;
                    end;
              end;
end;
```

procedure SimpleBPNet.backpropall(errorvec : pdynavec);

This procedure assigns errors to all neurons with input weights. On entry, errorvec contains the raw errors at the output layer, calculated somewhere else in the users code. Take care to pass the correct parameter to the derivative !

If you make a network with more than one hidden layer, you will probably want to call this method in your new backpropall method to take care of the last hidden layer. You would then write a very similar piece of code to calculate errors for your other hidden layers.

```
{---------------------------------}
procedure simpleBPnet.backpropall(errorvec    : pdynavec);
{---------------------------------}
var
    i           : integer;
    thisone   : pneuron;
    thiserror        : double;
    out               : double;
begin
                                  {scale raw errors for each output
neuron}

       for i := 1 to outputfield^.count do
                begin
                thiserror        := errorvec^.get(i);
                thisone           := outputfield^.at(i-1);
                out                := thisone^.output;
                thisone^.error := thisone^.dsfunc(out)*thiserror;
                end;

                                    {propagate back to the lower layers}

       backprop(outputfield,hiddenfield);
end;
```

procedure SimpleBPNet.getdeltaweights(lcoeff,momentum : double);

All errors are assigned; thus all changes in weights can be calculated. For each link, only the error at the destination neuron is needed. All weights are updated here.

```
{---------------------------------}
procedure simpleBPnet.getdeltaweights(lcoeff,momentum   : double);
{---------------------------------}
var
    i,j             : integer;
    source,dest   : pneuron;
    w,dw            : double;
    bumpup         : double;
    hiddenstart,
    outstart       : word;

procedure getdelta;
begin
          getneuron(j,dest);
          getneuron(i,source);
```

```
            dw          := deltaW^.get(i,j);
            bumpup      := lcoeff * (dest^.error) * (source^.output)
                              + momentum*dw;
            deltaw^.put(i,j,bumpup);
            Weights^.put(i,j,Weights^.get(i,j)+bumpup);
end;

begin
       hiddenstart := countin+1;
       outstart     := countin+counthidden+1;
       for i := 1 to countin do                        {loop from input
neurons}
          for j := hiddenstart to outstart-1 do     {to hidden neurons}
              getdelta;
       for i := hiddenstart to outstart-1 do       {loop from hidden
neurons}
          for j := outstart to count-1 do            {to output neurons }
              getdelta;
                           { Now do weights from offset neuron}
       i := count;
       for j := countin+1   to   count-1   do
             getdelta;

end;
```

procedure SimpleBPNet.train(errorvec : pdynavec);

Simply uses the errorvector to get all errors and calculates all weight changes.

```
{--------------------------------}
procedure simplebpnet.train(errorvec    : pdynavec);
{--------------------------------}
begin
       backpropall(errorvec);
       getdeltaweights(learn,momen);
end;
```

destructor SimpleBPNet.done; virtual;

Calls the inherited destructor and disposes of the DeltaW matrix.

```
{--------------------------------}
destructor simpleBPnet.done;
{--------------------------------}
begin
     neuralnet.done;
     dispose(deltaW,done);
end;
```

The Brain unit

On your distributiondisk, you should find the Borland units **BRAIN.TPU, BRAIN.TPP** and **BRAIN.TPW**, for DOS, DOS protected mode and Windows 3.1 respectively.

This unit provides an interface to the networks written by California Scientific's Brainmaker program. It contains 3 routines to extract data from Brainmaker .NET files, and uses them to implement a backprop network which scales its own input and output data to and from real-world values. Together, these tools allow you to quickly construct networks which give the same results as a 3 layer Brainmaker network with standard transfer functions.

The unit will only work with BP7, which provides the necessary string manipulation tools.

Source is provided.

The interface section

```
                                INTERFACE

uses objects,dyna2, cfmtools, strings, bpnet2, nnunit2;

type

 Pscalingbpnet = ^scalingbpnet;
 {------------------------}
 scalingbpnet   = object(SimpleBPnet)
 {------------------------}
     inputscale     : pdynavec;
     outputscale   : pdynavec;
     inputbuffer   : pdynavec;
     inmin,inmax,
     outmin,outmax : pdynavec;

     constructor init(incount,hiddencount,outcount : word;
                              lcoeff,momentum                  :
double);
     constructor load(var s : tstream);
     procedure store(var s : tstream);
     destructor   done;virtual;
     procedure getscaledata(var Brainfile : text);
     procedure scaleinputs( var inscales : dynavec);virtual;
     procedure scaleoutputs( var outscales : dynavec);virtual;
     procedure feedforward(datavec : pdynavec);virtual;
     procedure runthrough(input, result : pdynavec);
 end;
```

```
const

       RScalingBPnet : tstreamrec = (
              objtype   : 11410;
              vmtlink   : ofs(typeof(Scalingbpnet)^);
              load       : @scalingbpnet.load;
              store      : @scalingbpnet.store
       );


function BrainWeights(var thefile : text; mat    : pdynamat) : boolean;
function BrainNetSize(var thefile: text;
                      var inputs, hiddens, outputs : word): boolean;
function Brainscales (var thefile: text; inscales,outscales,
                                                    inmin,inmax,
                                                    outmin,outmax :
pdynavec) : boolean;
```

The Auxiliary functions

function BrainWeights(var thefile : text; mat : pdynamat) : boolean;

Reads a Brainmaker .NET file and extracts the weights into a single matrix, *mat*. The matrix (of type dynamat), must be the correct size before entry, i.e. NxN, where N is the number of neurons. A single offset neuron is assumed to be included in N. False is returned on any error. The file is already open on entry.

function BrainNetSize(var thefile: text; var inputs, hiddens, outputs : word): boolean;

Finds the size of a 3 layer Brainmaker net stored in *thefile*. Returns number of neurons in the input, hidden and output layers. *Thefile* should have been opened before entry to this routine. Returns true if successful.

function Brainscales (var thefile: text; inscales, outscales, inmin, inmax, outmin, outmax : pdynavec) : boolean;

Finds, in *thefile*, minimum and maximum value used for each input and each output neuron and returns this data in the last four parameters. Returns the scalefactors to be used for each input and output in *inscales* and *outscales*, i.e. multiplication of the data range by the scalefactor produces data in the range [0,1]. All dynavecs must be of correct size on entry. Note that only the scaling is

calculated, not the offset, which is dictated by the particular neuron's transfer function. Thus, *inmin, inmax, outmin, outmax* contain this data on exit.
File *thefile* is open on entry.

The ScalingBPNet object

This object is derived form the Simple BPNet object, and implements scaling of data presented to and produced by the network. A method for extracting scaling data from a Brainmaker .NET file is provided, but you can also use this object with scaling data generated elsewhere. The ScalingBPNet object thus acts on and produces 'real-world' data directly.

Note that you have to take some care if you want to train a ScalingBPNet after using the *runthrough* method - although this produces 'real-world' ouput in the result parameter, the output of each neuron is reset to its native value. When you calculate the error, and plan to use the inherited *train* method, take this into account.

ScalingBPNet data members

Inputscale A pointer to a Dynavec which contains the scale factors for the input data. Thus, the first entry in *Inputscale*^ is the number by which each 'real-world' input to neuron 1 in the input field must be multiplied after applying the offset but before entry into the network.

Outputscale A pointer to a Dynavec which contains the scale factors for the output data. Entry i scales the output of the i'th neuron in the output layer.

Inputbuffer A pointer to a Dynavec which buffers the input data (in the feedforward method) in order to apply an offset .

Inmin, inmax, outmin, outmax

Pointers to Dynavecs containing the minimum and maximum values over the whole training set.

ScalingBPnet methods

constructor scalingbpnet.init(incount, hiddencount, outcount : word; lcoeff, momentum : double);

Calls the inherited init method and instantiates the other data members.

```
begin
       inherited init(incount,hiddencount,outcount,lcoeff,momentum);
       new(inputscale,init(incount,1) );
       new(outputscale,init(outcount,1) );
       new(inputbuffer,init(incount,1) );
       new(inmin,init(incount,1));
       new(inmax,init(incount,1));
       new(outmin,init(outcount,1));
       new(outmax,init(outcount,1));
end;
```

constructor scalingbpnet.load(var s : tstream); This constructor loads the network from a stream. Calls the inherited method, then loads the datamembers in the order they were written.

```
begin
        inherited load(s);
        inputscale   := pdynavec(s.get);
        outputscale := pdynavec(s.get);
        inputbuffer := pdynavec(s.get);
        inmin    := pdynavec(s.get);
        inmax    := pdynavec(s.get);
        outmin      := pdynavec(s.get);
        outmax         := pdynavec(s.get);
end;
```

procedure scalingbpnet.store(var s : tstream);

Simply calls the inherited method, then writes the extra data members to the stream.

```
begin
        inherited store(s);
        s.put(inputscale);
        s.put(outputscale);
        s.put(inputbuffer);
        s.put(inmin);
        s.put(inmax);
        s.put(outmin);
        s.put(outmax);
end;
```

destructor scalingbpnet.done;

Disposes of the extra data members if necessary and calls the inherited destructor.

```
begin
```

```
      if inputscale <> nil then dispose(inputscale,done);
      if outputscale <> nil then dispose(outputscale,done);
      if inputbuffer <> nil then dispose(inputbuffer,done);
      if inmin <> nil then dispose(inmin,done);
      if inmax <> nil then dispose(inmax,done);
      if outmin <> nil then dispose(outmin,done);
      if outmax <> nil then dispose(outmax,done);
      inherited done;
end;
```

procedure scalingbpnet.getscaledata(var Brainfile : text);

A simple call to the auxilliary routine in this unit to get the scales and maxmin data from a Brainmaker file. The file should be open on entry.

```
begin
     Brainscales(Brainfile,
inputscale,outputscale,inmin,inmax,outmin,outmax);
end;
```

procedure scalingbpnet.scaleinputs(var inscales : dynavec);

Invec contains the scale factors for the input field. These are used to set the scalar field of each neuron in the input field. This routine assumes that the transfer function of all the input neurons is the *linear* function (all input layers should have this). The scaling is then done by the neuron itself when it calculates its transfer function. If you are training or running with 'real-world' data, you must call this routine before you begin.

```
var
    i    : integer;
begin
      for i := 1 to inputfield^.count do
          pneuron(inputfield^.at(i-1))^.setscale(inscales.get(i));
end;
```

procedure scalingbpnet.feedforward(datavec : pdynavec);

Applies the offset to the input data in *datavec* before feeding forward. Assumes that the input layer has the linear transfer function,and that scaling will be done by the neuron scalars, already set. Offsets the data in datavec so that the entry with lowest value becomes 0.0. Subsequent scaling when the input layer fires will then propagate data in the range [0,1] into the net.

```
var
    i    : integer;
begin

      for i := 1 to datavec^.count do
    inputbuffer^.put(i,
```

```
                        datavec^.get(i)-inmin^.get(i));
      inherited feedforward(inputbuffer);
end;
```

procedure scalingbpnet.scaleoutputs(var outscales : dynavec);

Changes the output of each output neuron to a 'real world' value by multiplying by the output scale factor and then adding the offset. Assumes a native output range of [0,1] for each neuron.The value that the neuron actually produced is still available in state.output, so firing it will get the native value back. Also see the *runthrough* method, which uses this.

```
var
    i               : integer;
    aneuron        : pneuron;
begin
      for i := 1 to outputfield^.count do
          begin
          aneuron := pneuron(outputfield^.at(i-1));
          aneuron^.output := aneuron^.output * outscales.get(i) +
                             outmin^.get(i);
          end;
end;
```

procedure scalingBPnet.runthrough(input, result : pdynavec);

Applies 'real-world' data in *input* to the network, and runs it through to the outputlayer and places the 'real-world' output into *result*. The outputs of the neurons in the output layer are restored to their native values by refiring the neurons in the output layer.

```
var
    i   : integer;
begin
      feedforward(input);
      scaleoutputs(outputscale^);
          {Get outputs after scaling and offset}
      for i := 1 to countout do   result^.put(i,
                                  pneuron(outputfield^.at(i-1))^.output);
                    {Get unscaled values back into outputs}
      for i := 1 to countout do
           pneuron(outputfield^.at(i-1))^.fire;
end;
```