NEURAL NETWORKS TOOLKIT

Version 1.0

17 CAPRI RD
ST JAMES
SOUTH AFRICA
December 1992

This Turbo Pascal unit was constructed with TP 6.0. It includes the basic tools necessary for the implementation of neural nets - to date, objects include neurons and nets of neurons. The unit uses the dynamic matrix unit, DYNA2, supplied with this unit. In order to make full use of this unit, you should understand the Tcollection object supplied with Turbo Pascal.

THE INTERFACE SECTION

```
CONST
    smallnumber        = 1.0e-9;
    neuralerrmsgcount  = 11;
    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 >>>'
                   );
```

```
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',
                            'dtanh',
                            'dhalfsine'  ,
                            'dsigmoid'   ,
                            '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}
       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 }
{-------------------------------------}

    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); virtual;
    procedure   getneuron(i : integer; var aneuron : pneuron); virtual;
    procedure   calcallstates; virtual;
    procedure   deleteneuron(var aneuron : pneuron);virtual;
    procedure   addfield(var field      : neuronfield;
                         startat, endat : integer);     virtual;
    procedure   deletefield(var field : neuronfield); virtual;
    procedure   fireall; virtual;
    procedure   killfield(var field   : neuronfield); virtual;
    procedure   getinputsof(thisone   : pneuron;
                            threshold : double;
                            var field : neuronfield);    virtual;
    procedure   presentinputto(thefield  : neuronfield;
                               thedata   : pdynavec);    virtual;
    procedure   connect(var f:neuronfield; weight: double);virtual;
    procedure   disconnect(var f:neuronfield);virtual;
    procedure   connectbetween(var from,into: neuronfield;
                               weight: double); virtual;
    procedure   disconnectbetween(var from,into: neuronfield); virtual;
    procedure   propagate; virtual;
    procedure   randomweights(alimit : double); virtual;
    procedure   nofeedback; virtual;
    procedure   setfieldsignal(var field : neuronfield;s : signaltype); virtual;
    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 fdtanh(tanhx: double) : double;
function fhalfsine(a: double): double;
function fdhalfsine(a: double): double;
function fstep(a: double): double;
function fsigmoid(a: double): double;
function fdsigmoid(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;           { Prints the current error
                                      message or nothing if all is well}
```

THE NEURON OBJECT

NEURON FIELDS

Sfunctype : Signaltype
Contains the signal funtion type used by the neuron. Signaltype is an enumerated type with possible values listed above.

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

<<<<<< **A note about setting up signalfunctions**: >>>>>>

When the neuron object is initialized, the addresses of the signalfunctions are resolved (see the 'Neuron Fields' section, where the code is given). As it stands, this unit ***makes an exeption*** for 2 cases where the derivative of the function is a simple function of the function value itself : here, the parameter passed to the derivative is the function value, not the activation. ***Watch out*** for this when you write gradient based training regimes. I did this because it speeds up training - the function value is already available by the time the derivative needs to be calculated. The transfer funtions in question are the SIGMOID function and the HYPERBOLIC TANGENT function. You can also do this for the GAUSSIAN function, but I didn't. I know this is a little messy, but you can easily change it if you want.
Here's the code :

```
{--------------------------------}
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 fdtanh(tanhx: double): double;
{--------------------------------}
begin
    fdtanh := (1.0-tanhx*tanhx);
end;


{--------------------------------}
function fsigmoid(a: double): double;
{--------------------------------}
begin
    fsigmoid := 1.0/(1.0 + exp(-a));
end;

{--------------------------------}
function fdsigmoid(sigx: double): double;  {a is fsigmoid(x) }
{--------------------------------}
begin
    fdsigmoid := 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}
{--------------------------------}
begin
    if (a > pi/2.0)
      then fdhalfsine := 1.0
    else
      if (a < -pi/2.0)
         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;

{--------------------------------}
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 := (@fdtanh);
      halfsine  : findsignalfunc := (@fdhalfsine);
      step      : findsignalfunc := (@fzero);
      sigmoid   : findsignalfunc := (@fdsigmoid);
      gaussian  : findsignalfunc := (@fdgaussian);
      one       : findsignalfunc := (@fzero);
      zero      : findsignalfunc := (@fzero);
```

```
        end;
    end;
```

(No attempt was made to optimize the buggers, and my first implementation of lookup tables failed to beat the 80387 - maybe you can help.)

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 neuronstate, so that it may remain hidden from the network until the algorithm requires a new output, and the neuron formally fires.

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

Error, Lasterror : double
Current and previous error. Used for some training algorithms.

NEURON METHODS

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

The neuron is initialized by specifying a signalfunction type and an initial neuronstate (see the constant 'reststate'). Calls setsignal, and sets scalar to 1.0.

procedure neuron.store(var s: tstream);

Writes the neuron to the stream s as follows : sfunctype, scalar, state, output, error, lasterror.

constructor neuron.load(var s: tstream);

Reads the neuron from the stream s by reading sfunctype, scalar, state, output, error, lasterror in this order, and then calling setsignal(sfunctype).

procedure 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. Setsignal looks 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;
```

procedure setscale (s : double);

Simply sets scalar to s.

procedure getstate(var s: Neuronstate);

Returns the current state of the neuron in s.

procedure 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 fire;

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

Destructor Done;

Nothing special here. Calls the ancestors done method.

THE NEURALNET OBJECT

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 a logical unit, 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 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 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 0.01 The fieldlist is created on the heap (init(3,1)) and inputfield and outputfield are set to NIL.

Constructor load(var s : tstream);

Loads the net from the stream s.

Procedure store(var s : tstream);

Stores the net on the stream s.

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

Makes a new neuron, adds it at position i in the net and adjusts 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 set to NIL on entry, and 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 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 should be NIL on entry and is NIL if i doesn't make sense.

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

On entry, field points to nothing. A field is initialized, neurons are inserted and the 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 or endat do not index neurons in the network, or startat > endat, nothing is done and an error is posted in neuralerror.

procedure 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. Field is NIL on exit if successfull. If field is not in fieldlist, nothing is done, and an error is posted in neuralerror.

procedure 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 posted in neuralerror.

procedure 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.

procedure 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 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 disconnect(var f:neuronfield);virtual;

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

procedure 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 the from field now propagates data to all neurons in the to field. 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 disconnectbetween(var from,into: neuronfield); virtual;

Calls connectbetween with a weight parameter of 0.0;

procedure propagate; virtual;

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

procedure 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 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 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 fireall ; virtual;

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

procedure 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. (The problem of sparseness remains largely unsolved - perhaps it is more properly addressed in a new 'backpropnet' object which definitely has a sparse weights matrix...)

destructor done; virtual;

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

```
{-------------------------- UNIT INITIALIZATION ----------------------}

begin
   neuralerror := 0;
   randomize;

     {Stream registration}

   registertype(Rneuron);
   registertype(Rneuralnet);

end.
```