**Simulation code**

The population models were run by integration c-shell scripts with the PlaNet5.7 neural network simulator

1. **Example c-shell script**

This shows the c-shell commands for training one network, specifying its initial parameters and saving performance metrics and weights each epoch. One thousand of these sets of commands were appended into a single extended script to run the population. A similar script for intervention loaded the weights and parameters from each individual, advanced to epoch 50, and then completed training using the full training set rather than the individual’s family training set.

This gives the script to train individual 1 (1 to 1000) in family 0 (0 to 499)

foreach o (0) # Family ID number

foreach a (10) # Number of hidden units

foreach b (0.75) # Sigmoid temperature

foreach c (0.5) # Processing noise

foreach d (0.175) # Learning rate

foreach e (0.2) # Momentum

foreach f (0.25) # Initial weight variance

foreach g (0) # Architecture (1=3-layer, 2=fully connected)

foreach h (1) # BP error metric (0=RMS, 1=cross-entropy)

foreach i (0.15) # Nearest neighbour response threshold

foreach j (100) # Epoch when pruning begins

foreach k (0.1) # Probability connection is pruned if below threshold

foreach l (0.5) # Pruning magnitude threshold

foreach m (0) # Connection weight decay rate

foreach n (0) # Sparseness of connectivity

foreach p (0.6) # Lower level of quotient for family training set

foreach q (1) # Upper level of quotient for family training set

foreach r (1) # Population ID number

# Generate family training set

RunNet

network Nin=58 Nhid1=75 Nout=62 n.env.net

pattern trainingsetnd2.pat

nset qualityl $p

nset qualityh $q

nset family\_ID $o

nset binary\_num 30

nset binary\_sparse 0.5

source generate\_family\_env.scr

print my\_quality >> database2\_family\_quality\_twin.out

quit

# Train individual

RunNet

savefile weights\_family.$o.individual.$r.wts

nset weight $f

network Nin=59 Nhid1=$a Nout=62 willnd2\_50h.net

nset floatpattern on

nset precision 7

nset eta 0.01

nset alpha 0

nset crosse $h

nset type $g

nset threshold $i

nset srate $m

nset temperature $b

nset Nose $c

exec initialise

nset lesion\_site 5

nset lesion\_level $n

exec prune

nset epoch 0

pattern phon\_my\_family\_$o.pat

nset eta $d

nset alpha $e

nset careoff 1

nset crosse $h

nset type $g

nset threshold $i

nset srate $m

nset temperature $b

nset Nose $c

nset lesion\_site 5

nset prune\_prob $k

nset prune\_threshold $l

nset prune\_epoch $j

save

/bin/cp blank bg\_phon\_train\_family.$o.individual.$r.out

/bin/cp blank connections.$r.out

foreach aloop (1 2 3 4 5 6 7 8 9 10)

foreach bloop (1 2 3 4 5 6 7 8 9 10)

 foreach cloop (1 2 3 4 5 6 7 8 9 10)

nset randomize on

pattern phon\_my\_family\_$o.pat

cycle 1

source test1.scr

exec prune\_small

cat results.out >> bg\_phon\_train\_family.$o.individual.$r.out

exec evaluate\_weights

print 'weights\_mag\_num' >> connections.$r.out

end

save

end

end

quit

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

end

1. PlaNet code for activating and updating the ANN

# File defines parameters for on-line measurement, then loads activation files and results matrices (codes\_nd2 = the codes for each phoneme in the triphonemic verbs)

define Ntrain 508 # number of training patterns

define Ntest 572 # number of testing patterns

define resize\_r 9 # size of the results matrix

define resize\_c 19 # (rows versus columns)

define phoneme\_r 32 # size of the phoneme codes matrix

define phoneme\_c 19 # (rows = number of phonemes, columns = units per phoneme)

define inflection\_r 4 # size of the inflection codes matrix

define inflection\_c 5 # (rows = number of inflections, columns = units per inflection)

load n.behaviour.genetic.act

load codes\_nd2

n.behaviour.genetic.act

#####################################################################

#

# Pruning network Activation file

#

# with temperature parameter, addition of noise, startstate pruning

# for sparseness, and ongoing pruning of small weights

#

#####################################################################

####################################################################

#

# Some definitions

#

####################################################################

## define Nin, Nhid1 and Nout as desired.

define ErrMsg \n\tread\swith\s'network\sNin=<no-of-input>\sNhid1=<no-of-hidden1>\sNhid2=<no-of-hidden2>\sNout=<no-of-output>\sn.3layer'\n

IFNDEF Nin; printf ErrMsg; exit; ENDIF

IFNDEF Nhid1; printf ErrMsg; exit; ENDIF

IFNDEF Nout; printf ErrMsg; exit; ENDIF

####################################################################

#

# Define layers, matrices, vectors, variables

#

####################################################################

## DEFINITIONS OF LAYERS

layer Input (Nin-1)

#layer Input (Nin-2) # CHANGE WHEN FIRST TWO BITS OF INPUT REQUIRED TO CODE COMMON ENVIRONMENT

layer Hidden1 Nhid1

layer Output Nout

define biasd user1

## DEFINITIONS OF LESION MATRICES

matrix InputHidden1\_les Nhid1 (Nin-1)

matrix InputOutput\_les Nout (Nin-1)

matrix Hidden1Output\_les Nout (Nhid1)

scalar countrows

scalar countcolumns

scalar lesion\_site

scalar lesion\_level

scalar lesion\_levelIND

scalar lesion\_levelDIR

scalar lrateIND 1

scalar lrateDIR 1

scalar prune\_prob # parameter that determines probability of cutting a small weight

scalar prune\_threshold # threshold magnitude, might cut a weight that is below this

scalar prune\_epoch # epoch after pruning is triggered

vector weights 3 # values for 3 layers, how many weights have been pruned

vector weights\_total 3 # total connections in layer prior to any lesioning

scalar temperature 1 # sets the slope of the sigmoid, 1=normal

scalar neuromodulation\_level # during aging, this is the proportion that temperature is reduced by each epoch

scalar age\_type # set to 1 for ageing via connection loss, set to 2 for ageing via neuromodulation (reduction in the temperature parameter)

## DEFINITIONS OF INPUT/TARGET BUFFERS

target Nout

input Nin

## WEIGHT DECAY RATE

scalar srate 0 # Allows proportional downscaling of weight matrix magnitudes during training (has no effect when 0)

## RANDOM NUMBER GENERATORS

# For noise procedures

scalar Nose 0 # noise added to layer vectors. This is the standard deviation of gaussian noise.

scalar rand

## DEFINITIONS OF CONNECTIONS

connect InputHidden1 Input to Hidden1

connect Hidden1Output Hidden1 to Output

connect InputOutput Input to Output

matrix connectivity1 sizeof(Hidden1) sizeof(Input)

matrix connectivity2 sizeof(Output) sizeof(Hidden1)

matrix connectivity3 sizeof(Output) sizeof(Input)

### DEFINITIONS OF WORKING VARIABLES

scalar crosse # 0 = bp with error, 1 = bp with cross entropy

scalar type # 0 = run 3-layer net, 1 = run 2-layer net, 2 = run combined net

scalar Freq # first bit of the input vector defines the frequency, 0->1 proportion of eta to apply.

scalar count # generic counter

scalar count2 # second generic counter

scalar countrows # counter for lesioning procedure

scalar countcolumns # counter for lesioning procedure

scalar patternnumber # indexes input pattern being tested

scalar traintest # indexes whether we are testing on training or generalisation set, for compiling results matrix

scalar threshold # error threshold below which nearest neighbour matches are acceptible, in one condition of the results matrix

scalar errorn # the error of the output to its nearest neighbour response

scalar middle 0.5 # used in careoff, downgrading error from zero outputs, for sparse localist training

scalar row # used in calculating entries of results matrix, determines pattern type

scalar Fre # frequency, 0=normal, 1=testing hf only, 2=testing lf only (for results averager)

vector indexerrorp phoneme\_r # used in nearest neighbour to decide closest phoneme

vector indexerrori inflection\_r # used in nearest neighbour to decide closest inflection

vector nearest\_out Nout # the identity of the nearest neighbour

scalar familybit 1

vector weights\_mag\_num 6 # to store summed weight magnitudes and number of remaining weights per layer (see procedure evaluate\_weights)

## Matrices storing candidate target outputs for each of the patterns

matrix results resize\_r resize\_c

matrix npercell resize\_r resize\_c

matrix npercellhf resize\_r resize\_c

matrix npercelllf resize\_r resize\_c

matrix tr\_reg Ntrain Nout

matrix tr\_irreg Ntrain Nout

matrix tr\_blend Ntrain Nout

matrix tr\_unmark Ntrain Nout

matrix gen\_reg Ntest Nout

matrix gen\_irreg Ntest Nout

matrix gen\_blend Ntest Nout

matrix gen\_unmark Ntest Nout

matrix phoneme\_codes phoneme\_r phoneme\_c

matrix inflection\_codes inflection\_r inflection\_c

## Macro definitions of the derivatives of the sigmoid for Hidden and Output

IF $min==0&&$max==1

define Hidden1Der Hidden1\*(1-Hidden1)\*temperature

define OutputDer Output\*(1-Output)\*temperature

ELSE

define Hidden1Der (Hidden1-$min)\*($max-Hidden1)/($max-$min)\*temperature

define OutputDer (Output-$min)\*($max-Output)/($max-$min)\*temperature

ENDIF

####################################################################

#

# Startstate pruning procedures

#

# Initialise. Specify lesion\_site and lesion\_level. Execute

#

####################################################################

procedure initialise

 connectivity1 = 1

 connectivity2 = 1

 connectivity3 = 1

 weights[0] = 0

 weights[1] = 0

 weights[2] = 0

 weights\_total[0] = sum(connectivity1)

 weights\_total[1] = sum(connectivity2)

 weights\_total[2] = sum(connectivity3)

end

procedure prune

 if lesion\_site==5 then # all connections evenly

 connectivity1=randomstate(connectivity1=(1-lesion\_level))

 connectivity2=randomstate(connectivity2=(1-lesion\_level))

 connectivity3=randomstate(connectivity3=(1-lesion\_level))

 InputHidden1=InputHidden1\*connectivity1

 Hidden1Output=Hidden1Output\*connectivity2

 InputOutput=InputOutput\*connectivity3

 endif

 weights\_total[0] = sum(connectivity1)

 weights\_total[1] = sum(connectivity2)

 weights\_total[2] = sum(connectivity3)

end

## Procedures for ageing the networks, not used in the SES populations

procedure prune\_age

 if lesion\_site == 5 then

 countrows = 0

 countcolumns = 0

 while countrows != Nhid1;

 while countcolumns != (Nin-1)

 if urandom(1) <= lesion\_level then

 connectivity1[countrows][countcolumns] = 0

 endif

 countcolumns = countcolumns + 1

 endwhile

 countcolumns = 0

 countrows = countrows + 1

 endwhile

 countrows = 0

 countcolumns = 0

 while countrows != Nout;

 while countcolumns != (Nhid1)

 if urandom(1) <= lesion\_level then

 connectivity2[countrows][countcolumns] = 0

 endif

 countcolumns = countcolumns + 1

 endwhile

 countcolumns = 0

 countrows = countrows + 1

 endwhile

 countrows = 0

 countcolumns = 0

 while countrows != Nout;

 while countcolumns != (Nin-1)

 if urandom(1) <= lesion\_level then

 connectivity3[countrows][countcolumns] = 0

 endif

 countcolumns = countcolumns + 1

 endwhile

 countcolumns = 0

 countrows = countrows + 1

 endwhile

 endif

 InputHidden1=InputHidden1\*connectivity1

 Hidden1Output=Hidden1Output\*connectivity2

 InputOutput=InputOutput\*connectivity3

 weights\_total[0] = sum(connectivity1)

 weights\_total[1] = sum(connectivity2)

 weights\_total[2] = sum(connectivity3)

end

procedure ageing # for neuromodulation, this reduces the temperature each epoch by a fixed proportion 'neuromodulation\_level'

 if age\_type == 1 then

 nset lesion\_site 5

 nset lesion\_level .003345

 call prune\_age

 endif

 if age\_type == 2 then

 nset neuromodulation\_level .01035

 temperature = temperature \* (1-neuromodulation\_level)

 endif

end

####################################################################

#

# Ongoing pruning procedures

#

####################################################################

procedure prune\_small

 if $epoch >= prune\_epoch then

 # This procedure tracks through each weight matrix, and sets each weight to zero

 # with a probability prune\_prob if its absolute value is less than the parameter

 # prune\_threshold. A matrix 'weight' updates how many weights have been set to zero

 #

 # prune\_prob: parameter that determines probability of cutting a small weight

 # prune\_threshold: threshold magnitude, might cut a weight that is below this

 #

 # connectivity1 is for InputHidden1

 # connectivity2 is for Hidden1Output

 # connectivity3 is for InputOutput

 #

 if lesion\_site == 5 then

 countrows = 0

 countcolumns = 0

 while countrows != Nhid1;

 while countcolumns != (Nin-1)

 if (((InputHidden1[countrows][countcolumns])^2)^0.5) < prune\_threshold then

 if urandom(1) <= prune\_prob then

 connectivity1[countrows][countcolumns] = 0

 endif

 endif

 countcolumns = countcolumns + 1

 endwhile

 countcolumns = 0

 countrows = countrows + 1

 endwhile

 countrows = 0

 countcolumns = 0

 while countrows != Nout;

 while countcolumns != (Nhid1)

 if (((Hidden1Output[countrows][countcolumns])^2)^0.5) < prune\_threshold then

 if urandom(1) <= prune\_prob then

 connectivity2[countrows][countcolumns] = 0

 endif

 endif

 countcolumns = countcolumns + 1

 endwhile

 countcolumns = 0

 countrows = countrows + 1

 endwhile

 countrows = 0

 countcolumns = 0

 while countrows != Nout;

 while countcolumns != (Nin-1)

 if (((InputOutput[countrows][countcolumns])^2)^0.5) < prune\_threshold then

 if urandom(1) <= prune\_prob then

 connectivity3[countrows][countcolumns] = 0

 endif

 endif

 countcolumns = countcolumns + 1

 endwhile

 countcolumns = 0

 countrows = countrows + 1

 endwhile

 endif

 # compute vector recording how many weights have been lesioned from each layer so far

 weights[0] = weights\_total[0] - sum(connectivity1)

 weights[1] = weights\_total[1] - sum(connectivity2)

 weights[2] = weights\_total[2] - sum(connectivity3)

 InputHidden1=InputHidden1\*connectivity1

 Hidden1Output=Hidden1Output\*connectivity2

 InputOutput=InputOutput\*connectivity3

 weights\_total[0] = sum(connectivity1)

 weights\_total[1] = sum(connectivity2)

 weights\_total[2] = sum(connectivity3)

 endif

end

procedure load\_connect

# for reinitialising a network’s full parameter and connect set when reloading trained weights

 countrows = 0

 countcolumns = 0

 while countrows != Nhid1;

 while countcolumns != (Nin-1)

 if InputHidden1[countrows][countcolumns] == 0 then

 connectivity1[countrows][countcolumns] = 0

 endif

 countcolumns = countcolumns + 1

 endwhile

 countcolumns = 0

 countrows = countrows + 1

 endwhile

 countrows = 0

 countcolumns = 0

 while countrows != Nout;

 while countcolumns != (Nhid1)

 if Hidden1Output[countrows][countcolumns] == 0 then

 connectivity2[countrows][countcolumns] = 0

 endif

 countcolumns = countcolumns + 1

 endwhile

 countcolumns = 0

 countrows = countrows + 1

 endwhile

 countrows = 0

 countcolumns = 0

 while countrows != Nout;

 while countcolumns != (Nin-1)

 if InputOutput[countrows][countcolumns] == 0 then

 connectivity3[countrows][countcolumns] = 0

 endif

 countcolumns = countcolumns + 1

 endwhile

 countcolumns = 0

 countrows = countrows + 1

 endwhile

 weights\_total[0] = sum(connectivity1)

 weights\_total[1] = sum(connectivity2)

 weights\_total[2] = sum(connectivity3)

end

####################################################################

#

# Activating the network and learning

#

####################################################################

## PROCEDURE FOR ACTIVATING NETWORK FORWARD

procedure activate

 if type==0 then # THREE LAYER

 if crosse==0 then # STANDARD BP

# Input=$input[1->(Nin-1)]

 Input=$input[2->(Nin-1)] # CHANGE WHEN FIRST TWO BITS OF INPUT REQUIRED TO CODE COMMON ENVIRONMENT

 Freq=$input[1]

 familybit=$input[0] # CHANGE WHEN FIRST TWO BITS OF INPUT REQUIRED TO CODE COMMON ENVIRONMENT

 Hidden1:net = InputHidden1\*\*T(Input)

 if Nose > 0 then

 call addnoise\_hidden1

 endif

 Hidden1=(1/(1+(exp(-(temperature\*(Hidden1:net+Hidden1:bias))))))

 Output:net = Hidden1Output\*\*T(Hidden1)

 if Nose > 0 then

 call addnoise\_output

 endif

 Output=(1/(1+(exp(-(temperature\*(Output:net+Output:bias))))))

 Output:delta=$target[0->(Nout-1)]-Output

 Output:delta = Output:delta\*(($target<middle)\*$careoff+($target>=middle))

 Output:delta \*= OutputDer

 endif

 if crosse==1 then # CROSS ENTROPY

# Input=$input[1->(Nin-1)]

 Input=$input[2->(Nin-1)] # CHANGE WHEN FIRST TWO BITS OF INPUT REQUIRED TO CODE COMMON ENVIRONMENT

 Freq=$input[1]

 familybit=$input[0] # CHANGE WHEN FIRST TWO BITS OF INPUT REQUIRED TO CODE COMMON ENVIRONMENT

 Hidden1:net = InputHidden1\*\*T(Input)

 Hidden1=(1/(1+(exp(-(temperature\*(Hidden1:net+Hidden1:bias))))))

 Output:net = Hidden1Output\*\*T(Hidden1)

 if Nose > 0 then

 call addnoise\_output

 endif

 Output=(1/(1+(exp(-(temperature\*(Output:net+Output:bias))))))

 Output:delta=$target[0->(Nout-1)]-Output

 Output:delta = Output:delta\*(($target<middle)\*$careoff+($target>=middle))

 endif

 endif

 if type==1 then # TWO LAYER

 if crosse==0 then # STANDARD BP

# Input=$input[1->(Nin-1)]

 Input=$input[2->(Nin-1)] # CHANGE WHEN FIRST TWO BITS OF INPUT REQUIRED TO CODE COMMON ENVIRONMENT

 Freq=$input[1]

 familybit=$input[0] # CHANGE WHEN FIRST TWO BITS OF INPUT REQUIRED TO CODE COMMON ENVIRONMENT

 Output:net = InputOutput\*\*T(Input)

 if Nose > 0 then

 call addnoise\_output

 endif

 Output=(1/(1+(exp(-(temperature\*(Output:net+Output:bias))))))

 Output:delta=$target[0->(Nout-1)]-Output

 Output:delta = Output:delta\*(($target<middle)\*$careoff+($target>=middle))

 Output:delta \*= OutputDer

 endif

 if crosse==1 then # CROSS ENTROPY

# Input=$input[1->(Nin-1)]

 Input=$input[2->(Nin-1)] # CHANGE WHEN FIRST TWO BITS OF INPUT REQUIRED TO CODE COMMON ENVIRONMENT

 Freq=$input[1]

 familybit=$input[0] # CHANGE WHEN FIRST TWO BITS OF INPUT REQUIRED TO CODE COMMON ENVIRONMENT

 Output:net = InputOutput\*\*T(Input)

 if Nose > 0 then

 call addnoise\_output

 endif

 Output=(1/(1+(exp(-(temperature\*(Output:net+Output:bias))))))

 Output:delta=$target[0->(Nout-1)]-Output

 Output:delta = Output:delta\*(($target<middle)\*$careoff+($target>=middle))

 endif

 endif

 if type==2 then # COMBINED

 if crosse==0 then # STANDARD BP

# Input=$input[1->(Nin-1)]

 Input=$input[2->(Nin-1)] # CHANGE WHEN FIRST TWO BITS OF INPUT REQUIRED TO CODE COMMON ENVIRONMENT

 Freq=$input[1]

 familybit=$input[0] # CHANGE WHEN FIRST TWO BITS OF INPUT REQUIRED TO CODE COMMON ENVIRONMENT

 Hidden1:net = InputHidden1\*\*T(Input)

 if Nose > 0 then

 call addnoise\_hidden1

 endif

 Hidden1=(1/(1+(exp(-(temperature\*(Hidden1:net+Hidden1:bias))))))

 Output:net = Hidden1Output\*\*T(Hidden1) + InputOutput\*\*T(Input)

 if Nose > 0 then

 call addnoise\_output

 endif

 Output=(1/(1+(exp(-(temperature\*(Output:net+Output:bias))))))

 Output:delta=$target[0->(Nout-1)]-Output

 Output:delta = Output:delta\*(($target<middle)\*$careoff+($target>=middle))

 Output:delta \*= OutputDer

 endif

 if crosse==1 then # CROSS ENTROPY

# Input=$input[1->(Nin-1)]

 Input=$input[2->(Nin-1)] # CHANGE WHEN FIRST TWO BITS OF INPUT REQUIRED TO CODE COMMON ENVIRONMENT

 Freq=$input[1]

 familybit=$input[0] # CHANGE WHEN FIRST TWO BITS OF INPUT REQUIRED TO CODE COMMON ENVIRONMENT

 Hidden1:net = InputHidden1\*\*T(Input)

 if Nose > 0 then

 call addnoise\_hidden1

 endif

 Hidden1=(1/(1+(exp(-(temperature\*(Hidden1:net+Hidden1:bias))))))

 Output:net = Hidden1Output\*\*T(Hidden1) + InputOutput\*\*T(Input)

 if Nose > 0 then

 call addnoise\_output

 endif

 Output=(1/(1+(exp(-(temperature\*(Output:net+Output:bias))))))

 Output:delta=$target[0->(Nout-1)]-Output

 Output:delta = Output:delta\*(($target<middle)\*$careoff+($target>=middle))

 endif

 endif

 $Error=mean(((($target-Output)^2)^0.5)) # the mean of the root of the squared error

end

### PROCEDURE FOR TRAINING NETWORK

procedure learn

 call activate

 if type==0 then # THREE LAYER

 Hidden1:delta = (Output:delta\*\*Hidden1Output) \* Hidden1Der

 InputHidden1 += InputHidden1:delta \

 = familybit\*Freq\*$eta\*lrateIND\*T(Hidden1:delta)\*\*Input + $alpha\*InputHidden1:delta

 InputHidden1 = InputHidden1 - (srate\*InputHidden1)

 Hidden1Output += Hidden1Output:delta \

 = familybit\*Freq\*$eta\*lrateIND\*T(Output:delta)\*\*Hidden1 + $alpha\*Hidden1Output:delta

 Hidden1Output = Hidden1Output - (srate\*Hidden1Output)

 Hidden1:bias += Hidden1:biasd = Hidden1:delta \* (familybit\*$eta\*Freq\*lrateIND) + Hidden1:biasd \* $alpha

 Output:bias += Output:biasd = Output:delta \* (familybit\*$eta\*Freq\*lrateIND) + Output:biasd \* $alpha

 endif

 if type==1 then # TWO LAYER

 InputOutput += InputOutput:delta \

 = familybit\*Freq\*$eta\*lrateDIR\*T(Output:delta)\*\*Input + $alpha\*InputOutput:delta

 InputOutput = InputOutput - (srate\*InputOutput)

 Output:bias += Output:biasd = Output:delta \* (familybit\*$eta\*Freq\*lrateDIR) + Output:biasd \* $alpha

 endif

 if type==2 then # COMBINED

 Hidden1:delta = (Output:delta\*\*Hidden1Output) \* Hidden1Der

 InputHidden1 += InputHidden1:delta \

 = familybit\*Freq\*$eta\*lrateIND\*T(Hidden1:delta)\*\*Input + $alpha\*InputHidden1:delta

 InputHidden1 = InputHidden1 - (srate\*InputHidden1)

 InputOutput += InputOutput:delta \

 = familybit\*Freq\*$eta\*lrateDIR\*T(Output:delta)\*\*Input + $alpha\*InputOutput:delta

 InputOutput = InputOutput - (srate\*InputOutput)

 Hidden1Output += Hidden1Output:delta \

 = familybit\*Freq\*$eta\*lrateIND\*T(Output:delta)\*\*Hidden1 + $alpha\*Hidden1Output:delta

 Hidden1Output = Hidden1Output - (srate\*Hidden1Output)

 Hidden1:bias += Hidden1:biasd = Hidden1:delta \* (familybit\*$eta\*Freq\*lrateIND) + Hidden1:biasd \* $alpha

 Output:bias += Output:biasd = Output:delta \* (familybit\*$eta\*Freq\*((lrateDIR+lrateIND)/2)) + Output:biasd \* $alpha

 endif

 # reset lesioned weights to zero

 InputHidden1=InputHidden1\*connectivity1

 Hidden1Output=Hidden1Output\*connectivity2

 InputOutput=InputOutput\*connectivity3

end

####################################################################

#

# On-line response measurement

#

# Generates the results matrix with nearest neighbour, thresholded

# nearest neighbour, error to nearest neighbour, and error to target

# values, in combination with the test.scr SCRIPT

# Results are for regular verbs and 3 types of irregular, then

# for novel verbs rhyming with each of the above + nonrhymes

#

####################################################################

procedure nearest

 # This option finds the closest phoneme match in each of the 3 phoneme slots, plus the closest

 # inflection match, then makes the nearest neighbour equal to these closest matches

 # (assumes phoneme\_codes matrix and inflection\_codes matrix have been loaded)

 # indexerrorp is a vector with the same number of elements as there are phoneme codes, ie phoneme\_r

 # indexerrori is a vector with the same number of elements as there are inflection codes, ie inflection\_r

 # nearest\_out is a vector with the same number of elements as there are output units, ie Nout

 # Nearest neighbour on first phoneme

 count=0

 while count!=(phoneme\_r);

 indexerrorp[count] = 1-mean(((phoneme\_codes[count]-Output[0->(phoneme\_c-1)])^2)^0.5)

 count = count + 1

 endwhile

 count = indexmax(indexerrorp)

 nearest\_out[0->(phoneme\_c-1)] = phoneme\_codes[count]

 # Nearest neighbour on second phoneme

 count=0

 while count!=(phoneme\_r);

 indexerrorp[count] = 1-mean(((phoneme\_codes[count]-Output[phoneme\_c->((2\*phoneme\_c)-1)])^2)^0.5)

 count = count + 1

 endwhile

 count = indexmax(indexerrorp)

 nearest\_out[phoneme\_c->((2\*phoneme\_c)-1)] = phoneme\_codes[count]

 # Nearest neighbour on third phoneme

 count=0

 while count!=(phoneme\_r);

 indexerrorp[count] = 1-mean(((phoneme\_codes[count]-Output[(phoneme\_c\*2)->((3\*phoneme\_c)-1)])^2)^0.5)

 count = count + 1

 endwhile

 count = indexmax(indexerrorp)

 nearest\_out[(phoneme\_c\*2)->((3\*phoneme\_c)-1)] = phoneme\_codes[count]

 # Nearest neighbour on inflection

 count=0

 while count!=(inflection\_r);

 indexerrori[count] = 1-mean(((inflection\_codes[count]-Output[(phoneme\_c\*3)->((phoneme\_c\*3)+inflection\_c-1)])^2)^0.5)

 count = count + 1

 endwhile

 count = indexmax(indexerrori)

 nearest\_out[(phoneme\_c\*3)->((phoneme\_c\*3)+inflection\_c-1)] = inflection\_codes[count]

end

# PROCEDURES to compare nearest neighbour with various possible responses and store

# various counts and errors in a matrix

procedure outcome\_tr

 # assumes relevant results matrices have been loaded

 # for each input to be tested, assign a patternnumber

 #

 # Section one: Is the NN equal to this response (count)

 # Section two: Is the NN equal to this response and the error less than the threshold (count)

 # Section three: If the NN is equal to this response, what is the error to the NN

 # Section four: What is the error to this response anyhow

 # Training set

 # regular row=0

 # irreg identity row=1

 # irreg vowel ch row=2

 # irreg arbit row=3

 #

 # targets (determines column)

 # 0,5,10,15: regular

 # 1,6,11,16: irregular

 # 2,7,12,17: blend

 # 3,8,13,18: unmarked

 # 4,9,14 : other

 if nearest\_out==tr\_reg[patternnumber] then # if the NN is a regular response

 results[row][0]=results[row][0]+1 # section one: increment NN count

 errorn=mean(((nearest\_out-Output)^2)^0.5)

 if errorn < threshold then

 results[row][5]=results[row][5]+1 # section two: increment thresholded-NN count

 endif

 results[row][10]=results[row][10]+errorn # section three: increment error to this NN

 else

 if nearest\_out==tr\_irreg[patternnumber] then # if the NN is irregular response .....

 results[row][1]=results[row][1]+1

 errorn=mean(((nearest\_out-Output)^2)^0.5)

 if errorn < threshold then

 results[row][6]=results[row][6]+1

 endif

 results[row][11]=results[row][11]+errorn

 else

 if nearest\_out==tr\_blend[patternnumber] then

 results[row][2]=results[row][2]+1

 errorn=mean(((nearest\_out-Output)^2)^0.5)

 if errorn < threshold then

 results[row][7]=results[row][7]+1

 endif

 results[row][12]=results[row][12]+errorn

 else

 if nearest\_out==tr\_unmark[patternnumber] then

 results[row][3]=results[row][3]+1

 errorn=mean(((nearest\_out-Output)^2)^0.5)

 if errorn < threshold then

 results[row][8]=results[row][8]+1

 endif

 results[row][13]=results[row][13]+errorn

 else

 results[row][4]=results[row][4]+1

 errorn=mean(((nearest\_out-Output)^2)^0.5)

 if errorn < threshold then

 results[row][9]=results[row][9]+1

 endif

 results[row][14]=results[row][14]+errorn

 endif

 endif

 endif

 endif

 results[row][15]=results[row][15] + mean(((tr\_reg[patternnumber]-Output)^2)^0.5) # section four: increment error to regular response

 results[row][16]=results[row][16] + mean(((tr\_irreg[patternnumber]-Output)^2)^0.5) # section four: increment error to irregular response ....

 results[row][17]=results[row][17] + mean(((tr\_blend[patternnumber]-Output)^2)^0.5)

 results[row][18]=results[row][18] + mean(((tr\_unmark[patternnumber]-Output)^2)^0.5)

end

procedure outcome\_gen

 if nearest\_out==gen\_reg[patternnumber] then

 results[row][0]=results[row][0]+1

 errorn=mean(((nearest\_out-Output)^2)^0.5)

 if errorn < threshold then

 results[row][5]=results[row][5]+1

 endif

 results[row][10]=results[row][10]+errorn

 else

 if nearest\_out==gen\_irreg[patternnumber] then

 results[row][1]=results[row][1]+1

 errorn=mean(((nearest\_out-Output)^2)^0.5)

 if errorn < threshold then

 results[row][6]=results[row][6]+1

 endif

 results[row][11]=results[row][11]+errorn

 else

 if nearest\_out==gen\_blend[patternnumber] then

 results[row][2]=results[row][2]+1

 errorn=mean(((nearest\_out-Output)^2)^0.5)

 if errorn < threshold then

 results[row][7]=results[row][7]+1

 endif

 results[row][12]=results[row][12]+errorn

 else

 if nearest\_out==gen\_unmark[patternnumber] then

 results[row][3]=results[row][3]+1

 errorn=mean(((nearest\_out-Output)^2)^0.5)

 if errorn < threshold then

 results[row][8]=results[row][8]+1

 endif

 results[row][13]=results[row][13]+errorn

 else

 results[row][4]=results[row][4]+1

 errorn=mean(((nearest\_out-Output)^2)^0.5)

 if errorn < threshold then

 results[row][9]=results[row][9]+1

 endif

 results[row][14]=results[row][14]+errorn

 endif

 endif

 endif

 endif

 results[row][15]=results[row][15] + mean(((gen\_reg[patternnumber]-Output)^2)^0.5)

 results[row][16]=results[row][16] + mean(((gen\_irreg[patternnumber]-Output)^2)^0.5)

 results[row][17]=results[row][17] + mean(((gen\_blend[patternnumber]-Output)^2)^0.5)

 results[row][18]=results[row][18] + mean(((gen\_unmark[patternnumber]-Output)^2)^0.5)

end

procedure reseter # resets results matrix to re-accumulate scores

 results = 0

end

value npercellhf

205 205 205 205 205 205 205 205 205 205 205 205 205 205 205 205 205 205 205

 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10

 34 34 34 34 34 34 34 34 34 34 34 34 34 34 34 34 34 34 34

 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5

410 410 410 410 410 410 410 410 410 410 410 410 410 410 410 410 410 410 410

 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20

 76 76 76 76 76 76 76 76 76 76 76 76 76 76 76 76 76 76 76

 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10

 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56

value npercelllf

205 205 205 205 205 205 205 205 205 205 205 205 205 205 205 205 205 205 205

 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10

 34 34 34 34 34 34 34 34 34 34 34 34 34 34 34 34 34 34 34

 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5

410 410 410 410 410 410 410 410 410 410 410 410 410 410 410 410 410 410 410

 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20

 76 76 76 76 76 76 76 76 76 76 76 76 76 76 76 76 76 76 76

 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10

 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56

procedure averager

# This procedure divides each cell of results matrix by the number of data

# points in that cell. For the counts, a proportion is generated (ie

# proportion correct, or proportion this particular response. For the error,

# an average error is generated.

# When the average error is to be calculated to the nearest neighbour for each

# of the response type \*only when that response type is the nearest neighbour\*,

# the total error has to be divided by the number of times that response turned

# out to be the nearest neighbour. However, first we have to check that number

# is greater than zero, otherwise we end up dividing by zero....

 count=0

 count2=10

 while count2!=(15)

 while count!=(resize\_r);

 if results[count][(count2-5)]!=0 then

 npercell[count][count2]=results[count][(count2-10)]

 # NB you want to divide the cumulative nearest neighbour

 # error by the count of correct nearest neighbour to work out the average error

 # per nearest neighbour.

 npercellhf[count][count2]=results[count][(count2-10)]

 npercelllf[count][count2]=results[count][(count2-10)]

 endif

 count = count + 1

 endwhile

 count2 = count2 + 1

 endwhile

 if Fre==0 then

 results = results/npercell

 endif

 if Fre==1 then # high frequency

 results = results/npercellhf

 endif

 if Fre==2 then # low frequency

 results = results/npercelllf

 endif

end

####################################################################

#

# Noise procedures

#

####################################################################

procedure random

 rand = urandom(1) # urandom(x) generates random numbers with uniform distribution in range [0,x]

end

procedure addnoise\_output

 noise Output:net Nose # variance of gaussian noise added to vector elements. Set to e.g. 0.2

# noise Output Nose

# Output = bound01(Output) # bound01() limits values between 0.0 and 1.0

end

procedure addnoise\_hidden1

 noise Hidden1:net Nose # variance of gaussian noise added to vector elements. Set to e.g. 0.2

# noise Hidden1 Nose

# Hidden1 = bound01(Hidden1)

end

vector neat 6

vector neatold 6

procedure doneat

 neat[0]=$epoch

 neat[1]=results[0][5]

 neat[2]=results[1][6]

 neat[3]=results[2][6]

 neat[4]=results[3][6]

 neat[5]=results[4][5]

 neatold[0]=$epoch

 neatold[1]=results[0][0]

 neatold[2]=results[1][1]

 neatold[3]=results[2][1]

 neatold[4]=results[3][1]

 neatold[5]=results[4][0]

end

procedure evaluate\_weights

 weights\_mag\_num[0]=sum(((InputHidden1)^2)^0.5) # IH summed magnitudes

 weights\_mag\_num[2]=sum(((Hidden1Output)^2)^0.5) # HO summed magnitudes

 weights\_mag\_num[4]=sum(((InputOutput)^2)^0.5) # IO summed magnitudes

 # count how many non-zero IH weights are left

 weights\_mag\_num[1]=0

 countrows = 0

 countcolumns = 0

 while countrows != Nhid1;

 while countcolumns != (Nin-1)

 if InputHidden1[countrows][countcolumns] != 0 then

 weights\_mag\_num[1]= weights\_mag\_num[1] + 1

 endif

 countcolumns = countcolumns + 1

 endwhile

 countcolumns = 0

 countrows = countrows + 1

 endwhile

 # count how many non-zero HO weights are left

 weights\_mag\_num[3]=0

 countrows = 0

 countcolumns = 0

 while countrows != Nout;

 while countcolumns != (Nhid1)

 if Hidden1Output[countrows][countcolumns] != 0 then

 weights\_mag\_num[3]= weights\_mag\_num[3] + 1

 endif

 countcolumns = countcolumns + 1

 endwhile

 countcolumns = 0

 countrows = countrows + 1

 endwhile

 # count how many non-zero IO weights are left

 weights\_mag\_num[5]= 0

 countrows = 0

 countcolumns = 0

 while countrows != Nout;

 while countcolumns != (Nin-1)

 if InputOutput[countrows][countcolumns] != 0 then

 weights\_mag\_num[5] = weights\_mag\_num[5] + 1

 endif

 countcolumns = countcolumns + 1

 endwhile

 countcolumns = 0

 countrows = countrows + 1

 endwhile

end