CS2NN16 Neural Networks : Part B

In Part B of the course,

I build on the single layer perceptrons already discussed

Covering Multi-Layer perceptrons

Including learning algorithm and Programming

Give Case Studies

Give alternative learning algorithms

Including an introduction to Genetic Algorithms

Then consider Radial Basis Functions

Finally discuss Weightless Neural Networks

Including Stochastic Diffusion Search

6 : Implementing MLP and Data

This lecture will address two issues

a) how to complete the implementation of a MLP

b) practical issues surrounding the use of an MLP

For a) we will outline the extra class needed to extend the single layer network classes to multi-layer

You will fill in the details in the assignment.

For b) we will consider practicalities associated with using the program – data processing; size of network; how to tell when to stop learning, etc.

This should be useful when you apply your network to a real world application.

Implementing MLP

First we will describe how the existing classes can be extended to implement a Multi-Layer Perceptron having

Hidden neurons with sigmoid activation

Output neurons with linear or sigmoidal

We already have classes for complete networks

LinearLayerNetwork	layer of linear activated neurons
SigmoidalLayerNetwork	layer of sigmoidal neurons

We will extend to have

MultiLayerNetwork	layer of hidden sigmoidal neurons

this craftily also has link to the output layer

and hence is a multi layer network

Class MultiLayerNetwork

This class should inherit from SigmoidalLayerNetwork

Data wise : it will have a pointer to the output layer

It will need a constructor (calling that for SigmoidalLayerNetwork)

and which initialises the output layer pointer

It will need a destructor – to delete the output layer – and itself

It will need its own version of many of the functions – mainly these

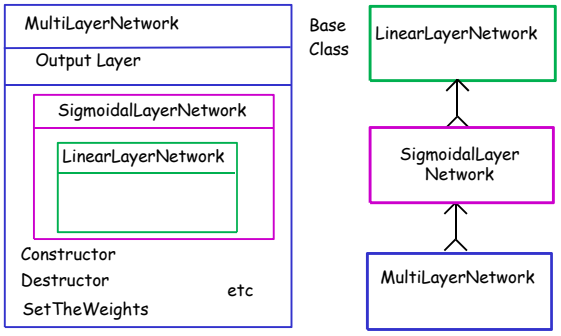
will process the neurons in its layer and those in the output layer.

To help the implementation of this, given below are

part of the class declaration

the constructor, ComputeNetwork

Inheritance



Part of Class Declaration

```
class MultiLayerNetwork : public SigmoidalLayerNetwork {
 LinearLayerNetwork *nextlayer;    // link to next layer
  virtual void CalcOutputs (vector<double> ins);
public:
  MultiLayerNetwork (int numIns, int numOuts,
                     LinearLayerNetwork *tonextlayer); // constructor
  virtual void SetTheWeights (vector<double> InitWt);
}
```

Create network with 2 inputs, 2 hiddens and 1 output by

```
net = new MultiLayerNetwork (2, 2,
                             new SigmoidalLayerNetwork (2,1) );
```

Pass pointer to output layer as third argument.

Constructor and Destructor

```
MultiLayerNetwork::MultiLayerNetwork (int numIns, int numOuts,
                    LinearLayerNetwork *tonextlayer)
                        :SigmoidalLayerNetwork (numIns, numOuts) {
                    // call inherent constructor to set up this layer
    nextlayer = tonextlayer;     // then set so point to next
}
MultiLayerNetwork::~MultiLayerNetwork() {
          delete nextlayer;
          // remove output layer, then auto-call inherited destructor
}
```


Doing these functions

```
void LinearLayerNetwork ::CalcOutputs(vector<double> ins) {
          // calc outputs of network given the inputs ins
    SigmoidalLayerNetwork::CalcOutputs(ins);
          // call hidden layer CalcOutputs using ins
    nextlayer -> CalcOutputs (outputs);
          // calc outputs of next layer : inputs being outputs
}
```

ChangeAllWeights also processes current and next layers
So has lines like the following

```
    nextlayer -> ChangeAllWeights ( … )    // do output layer
     SigmoidalLayerNetwork :: ChangeAllWeights ( … ) // hidden layer
```


Other Functions in SigActHidLayer

```
void MultiLayerNetwork::SetTheWeights (vector<double> initWt) {
   vector<double> wthis(initWt.begin(), initWt.begin() + numWeights);
   SigmoidalLayerNetwork::SetTheWeights(wthis);
   vector<double> wrest(initWt.begin() + numWeights, initWt.end());
   nextlayer -> SetTheWeights(wrest);
}                                  // and rest for output layer
```

SetTheWeights is passed weights of whole net,
first few weights initialise current layer, rest are for nextlayer
Achieved by defining subvectors using the begin() and end()

To do ReturnTheWeights vector.insert may be useful

On Finding Errors

Here need to

- Find errors and thence deltas in output layer
- Then find errors in hidden layer
- These are weighted sum of deltas in the output layer
  - ie need weights and deltas of the output layer
  - these are all in output layer
- So, add a member function in LinearLayerNetwork
  - passed the hidden layer's errors array
  - function calcs errors and stores them in this
- Then call FindDeltas function so 'errors' → deltas

Debugging

In last week's lecture values were given for deltas, deltaweights and weights when the XOR data were presented

(starting with 'Picton' weights).

If your network does not seem to work properly, then you should add to your program code to print out values, which you can compare with those.

May be useful to write member functions to return values

```
vector<double> getdata ();
```

this could put in data weights, deltas, deltaweights

could be output using the arrout function in the datasets library

Data Issues

In principle, your classes will be adaptable to a network with any number of inputs, hidden neurons and outputs.

You could thus change the main program so you could use the network for any specific application.

However, you can't just 'throw' data at a network & expect it to work – there are various considerations to be made

- How many inputs are there – you may not want to use all
- How many hidden neurons – how do you decide
- Is there any data pre-processing needed?
- Or post-processing?

Let us investigate

Under-fitting and Over-fitting

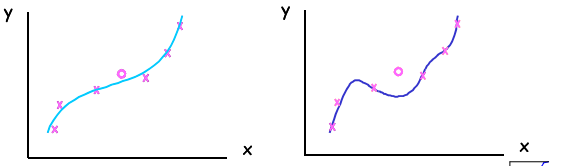
A Neural Network can model data – not exactly.

It must be detailed enough to not only model the training data, but also accurately 'predict' other data

If too simple wont learn training set - underfitting

A network too complex may fit data 'noise' – overfitting

Below predict y, train on x, unseen o



Network Size

How many neurons in hidden layer?

How many hidden layers?

If too many relative to training set → over-fitting

No formal methods known to solve this:

'folklore' includes Kolmogorov

'One hidden layer can learn any continuous function'

Two layers can improve matters sometimes

Later we show how Genetic Algorithms can help

When to Stop Learning

Train ANN to model training data but use on unseen data

As keep learning, SSE on training data down

If learn too much

network fits intricate details (or noise) of training data (overfits).

Errors on unseen data up.

Must stop training before then

How decide?

Training, Validation & Unseen Sets

Ans: have a data set independent of training set - validation

Thus have three data sets

Training set

Examples used for learning, i.e. to fit paras [weights].

Validation set

Examples used to tune the parameters of a classifier,

to decide when to stop learning

or to choose number of hidden units in a network

Unseen or Test set

Examples used only to assess the actual performance [generalization] of a fully-specified classifier.

How to Use Validation Set

```
Set prevValidSSE to high number; Learnt = false
while (not Learnt) and (numepochs < somemax) {
    Pass training set to network, adjust weights
    if (numepochs > suitable number)
        Pass validation set to network, calculate SSE
        if (newValidSSE > previousValidSSE)
            Learnt = true
        else
            remember new SSE
}
```

NB generally better to average ValidSSE over 10 epochs

On Input Data

Data sets may have many variables:

which affects output(s) the most?

Too many inputs slow learning or → overfitting

One can try using different inputs, train a network and see how well it performs (using the unseen data)

But, requires many experiments run and evaluated

Can use 'expert' knowledge about the problem,

or Mutual Information,

Principal Component Analysis,

Kohonen nets

These can allow you to extract less useful variables

Can be useful to add Extra Data

Can also be useful to create extra variables –

Suppose time of day used being 0 to 23 for hours.

Possible that output at 23.00 similar to that at 0.00

So introduce cyclic dummy variables

sin(time*2*π)/24

and

cos(time*2*π)/24

Need both

Can similarly apply to months of year / days of year

Pre-Processing

One point to consider is the range of input variables

If for instance, one variable, a, varied 0..100 and another, b, 0..2, and Euclidean distance between two a and two b variables was to be found, $d = \sqrt{((a_1-a_2)^2 + (b_1-b_2)^2)}$

Likely that the distance between two a's would be much greater than between two bs and so swamp the b's.

To solve this it is best to 'normalise' –

e.g. to scale data so in range 0..1 or -1 to +1

or to use statistical measures of the data to spread the data out and remove outliers

This not so crucial for MLPs … essential for some networks

Normalisation

Let *Dmin* and *Dmax* be min and max of a data variable

Let input range required for the network be *Imin* to *Imax*

Then to transform each data value *D* to an input value I is:

*I = Imin + (Imax - Imin) * (D - Dmin) / (Dmax - Dmin)*

Or, for the more statistically minded

*Dmin = mean (data) – k * std (data)* (k = 2 is ok)

*Dmax = mean (data) + k * std (data)*

and then compute I from D with these values

Possible (sometimes good) to use different normalisation methods for different variables. But same method(s) should be used on training, validation & unseen data sets

Post-Processing

One point worth noting is that it may be appropriate to post-process the network outputs.

For instance, consider an MLP with sigmoid activation

By definition the output of this will be in the range 0..1

Suppose application has outputs in the range 0..100, then there will need to be some scaling of the outputs

Also to be remembered, of course, is that the expected outputs in the training/validation/unseen sets will have to be scaled down to the range 0..1

Fortunately for you … the datasets class handles this…

Don't Use It Once

MLPs are used to generate non linear models

The learning process will mean the network will move (hopefully downwards) in the weight error space.

The starting position affects the end position, and whether there are local minima where the net may be trapped.

Thus you cant rely on running the network once.

Better to run many times, from different starting points

Then use the best result

Or average over a few results – a technique called bagging

Next week we will look at a test case, and consider alternatives to back propagation.

Assignment – plan implementation of MultiLayerNetwork…

7 : MLP Case Study and Alternatives

In this lecture we cover

a case study,

two alternatives to Back Prop

Chemotaxis and Directed Random Search

and introduce Genetic Algorithms

and show how they can be used in MLPs.

Case Study

This brief study is to predict Electricity Demand

based on data comprising

temperature, illumination and demand

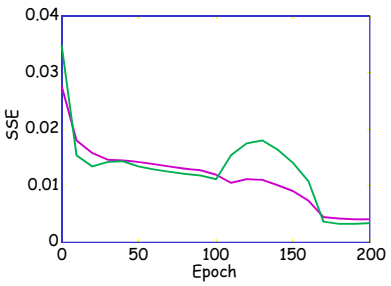
There are 168 data sets for training (7 days); 72 sets (3 days) for validation; and 24 sets (1 day) for (unseen)

The demand values are in the range approximately 18 to 40, not achievable with sigmoidal output, so scaled

scaled data = 0.1 + 0.8 * (data – min) / (max – min)

Network: 2 input neurons, 10 hidden neurons and 1 output Trained until the validation error averaged over 10 epochs rose .. test initiated after 150 epochs.

SSE of Train & Valid vs Epoch



Stops when after 200 epochs

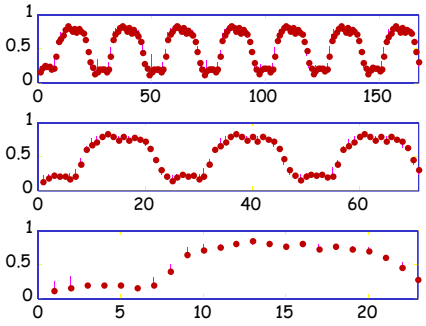
SSE train = 0.00409

SSE valid = 0.00331

SSE unseen = 0.0048

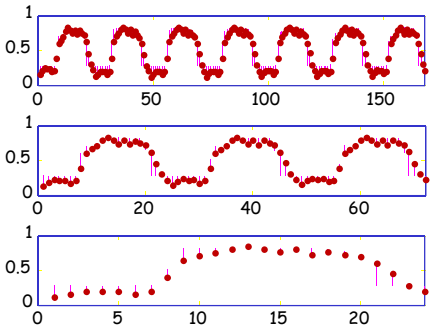
Blip in ValidSSE is reason for delaying stopping test

Tadpole Plot



Plots of actual (as dots) + lines from estimates to the dots; for training, validation and unseen data

Tadpole Plot when 5 Hidden Neurons



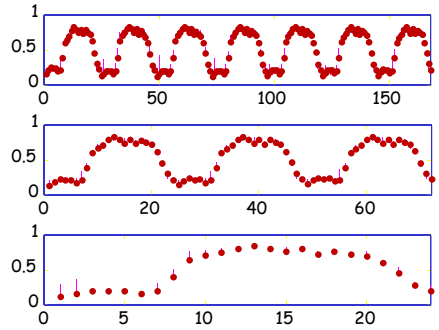
Training error is 0.01134,

Valid error 0.01135

Unseen error 0.01264

After 330 epochs

And When 30 Hidden Neurons



Training error is 0.00417,

Valid error is 0.00265

Unseen error is 0.00602

After 220 epochs

Summary

We have tried different numbers of hidden neurons

Of these tests, having 10 hidden neurons is the best

– 5 neurons is too few, 30 led to overfitting.

Note, as start with random weights, you cannot say from this one test that this is true.

Should do at least 10 tests on each & note average error.

Should also do with different learning rate/momentum.

Better to have more inputs: time, sin(time) etc.

NOTE these when you are doing assignment on the same example data, and of course on your application.

You can now finish the first part of the assignment : adapt the main program to use three sets of linear data.

Alternatives to Back Propagation

The delta rule, as it involves the differential of a node's output, is a calculus based learning rule.

A stochastic learning rule has some 'random' movements (though not completely random) – examples include Chemotaxis and Directed Random Search (DRS)

As random, time taken can vary for the same data.

For 'compact' data set, stochastic guaranteed to find the global minimum weight-error value.

If weight-error surface smooth, calculus prob more efficient.

However, time for one epoch for random << calculus.

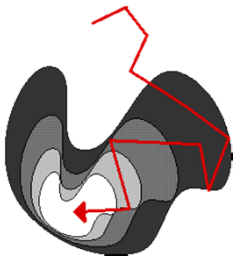
Let us now look briefly at these two stochastic methods.

Chemotaxis –Bacteria Movement

Bacteria move in same direction until food concentration stops going up, when they try a new direction selected at random.

By moving in a constant direction while concentration increases, Chemotaxis applies a 'directed component' on to a simple 'random walk'.

Bremermann HJ & Anderson RW (1989) Technical Report PAM-483 UC Berkeley



Used in recent papers on ANNs for finance …

The Algorithm

Initialise weights to small random values;
Calc objective func E_k.
while (network not trained ok)
  Generate ΔW_k from Gaussian distribution (mean 0, std 1)
  do
    Tentatively modify weight vector: $t_k = W + H \Delta W_k$
    Re-calculate objective function, E_{k+1}
    if ($E_{k+1} < E_k$) $W = t_k$; // if successful, adopt it
  while ($E_{k+1} < E_k$)

Automatic Adjustment of H – to speed up
  If num consecutive successful moves = n, H := H * Hfac
  If num consecutive failures = n, H := H / Hfac
    (when change H, reset consecutive count)

Directed Random Search

Ref Baba N (1989) Neural Networks **2:5** pp367-373

DRS is very similar to Chemotaxis ...
  Small random weight vector added to current weights
  Objective function re-evaluated
  If change improves performance it is made permanent.

The Random Step (The Forward Step)
  initial weight vector of uniformly distributed numbers
  Delta weight steps (ΔW_k) from mean 0 rand distribution
    biased by a directed component vector, ΔC_k.
  As usual, H, controls the size of the step in weight space.

The Reversal Step
  Applied if Forward Step doesn't improve overall error E_k.
  Geometrically analogous to looking in opposite direction..

The Directed Component Vector

Directed components provide an impetus or direction to search (analogous to momentum in Delta Rule)

Reflect history of success / failure for previous steps.

Initialised to zero

Added to random components at each step.

Using directed component provides a dramatic performance improvement to the random step or random step with reversals techniques: $W_{k+1} = W + \Delta W_k + \Delta C_k$

Hence successful steps (either forward or reversal) receive reinforcement via the directed component bias to the random perturbation vector.

Unsuccessful steps force this bias to be scaled back.

DRS Algorithm

W = RND: ΔC_0 = [0.0] // Initialise: RND -> next rand num
ΔWk = RND // Generate new direction to search
while (network not trained ok)
  $t_k = W + H\Delta W_k + \Delta C_k$ // Try forward step in this dirn
  E_{k+1} calculated
  if ($E_{k+1} < E_k$) // Forward step successful
    $W = t_k$; $\Delta C_{k+1} = 0.2 \times \Delta C_k + 0.4 \times \Delta W_k$;
  else $t_k = W - H\Delta W_k$ // Try reverse step
    E_{k+1} calculated
    if ($E_{k+1} < E_k$) // if Reverse step successful}
      $W = t_k$; $\Delta C_{k+1} = \Delta C_k - 0.4 \times \Delta W_k$
    else $\Delta C_{k+1} = 0.5 \times \Delta C_k$; ΔWk = RND
H can be adjusted in similar way to Chemotaxis

Genetic Algorithms – for MLPs

A Genetic Algorithm (GA) is a search technique

The aim is to find the best solution to a problem

GAs are inspired by evolutionary processes

Hence you also see the term Evolutionary Computation

GAs can be used for various tasks, but specifically for ANNs, it has been suggested that a GA could

find best network architecture for a given task (ok)

find the weights of an ANN (poor)

select the best input data

Note – although potentially useful in ANN design, they are often very computationally expensive – though are used

Basic Idea

Start with initial 'population' of possible solutions. Then

REPEAT

calculate the 'fitness' of each in population

use (probabilistically) better members of population to 'breed' next generation of the population

incorporate 'children'

UNTIL problem solved, or done enough iterations

How to represent population ? simplest case a series of bits

How to calculate fitness ? problem dependent

How to 'breed' ? need suitable genetic operators

How to incorporate next generation ? replace weakest or all

Encoding Populations &Fitness

Consider using a GA to set the architecture of an MLP, with 3 hidden layers with up to 15 neurons in each layer. Encode MLP's architecture as 'chromosome genotype' in binary

1	1	0	1	0	0	0	1	1	1	0	0
	L	1			L	2			L	3	

Genetic search bounded 4 bits - max 15 neurons per layer.
There are n such chromosomes
Fitness of each obtained by configuring an MLP for each, training it as usual, and looking at the error on unseen set.
Or, to select which of 6 inputs & num (< 32) in hidden layer

1	0	1	1	0	1	0	1	1	0	1
v_1	v_2	v_3	v_4	v_5	v_6					

Weighted Roulette Wheel Selection

Genetic optimisation is done by 'breeding' from best parents

Select with some randomness, but more likely to choose best.

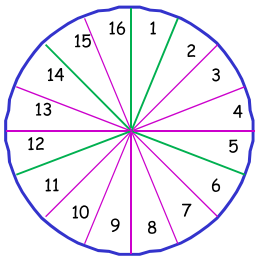
Suppose 5 in population, fitnesses (f) of 1, 4, 6, 3 and 2 (total 16)

Form wheel with 16 slots, allocating f slots per 'parent'

Get random number in range 1..16

Choose parent associated with slot

More likely to choose 'parent' 3



If R = 4, choose parent 2

Breeding – On 1 or 2 Chromosomes

Inversion – swap 2 random bits in chromosome

1	1	0	1	0	0	0	1	1	1	0	0		1	1	0	0	0	0	1	1	1	1	0	0
			↑			↑																			

Mutation – invert one random bit (use with low prob.)

1	1	0	1	0	0	0	1	1	1	0	0		1	1	0	1	1	0	0	1	1	1	0	0
				↑																					

Crossover – combine parts of two (? split at boundaries)

1	1	0	1	0	0	0	1	1	1	0	0
				↕							
0	0	1	0	1	1	1	0	0	1	1	0

1	1	0	1	0	1	1	0	0	1	1	0

Next Generation

Simple strategy - Kill (say) two with worst fitness

Breed 2 new ones (using the operators) to replace them

Then calculate their fitness

Suppose population fitnesses were

1 4 6 3 2 4

The 1st & 5th are killed off. Suppose now the fitnesses are

3 4 6 3 5 4

The overall fitness has now been increased.

Over time Darwinian evolution ensures survival of fittest, and hence over time, best solution to the problem found.

There is much more to GAs – this is a brief summary and shows how could be used for MLPs .. & that ends MLPs.

Next week Radial Basis Functions …

8 : Radial Basis Function Networks

As we have seen, one of the most common types of neural network is the multi-layer perceptron

It does, however, have various disadvantages, including the slow speed in learning

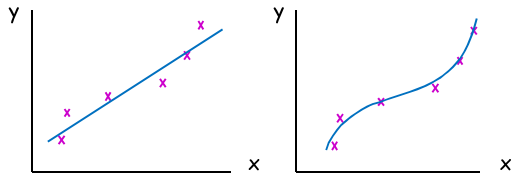
In this lecture we will consider an alternative type

The Radial Basis Function (or RBF) network

See Broomhead DS and Lowe D, 1988, Multivariable functional interpolation and adaptive networks, Complex Systems, 2, 321-355.

First we will make a note of fitting models

Fitting Data Points



In the first figure, a line is set as the best approx for the data (can be done by minimising the square of errors)

A better solution is a curve: could be done by a series of lines

Each line at an 'operating point' – this leads to RBF nets

If 3D have planes, in more dimensions have hyperplanes

Centres

Basic idea is that there are 'centres'

At each centre there is a simple model

This model is a 'basis' function which processes the input data according to its 'distance' from the associated centre

Each centre is like a neuron

The basis function is similar to a MLP's activation function

The output(s) are the weighted sum of the output of each 'neuron'

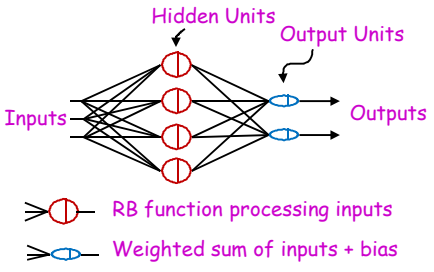
Overall, a RBF network has

an input layer, comprising the inputs to the network

a hidden layer, comprising the 'basis' function neurons,

an output layer, doing weighted sum of hidden layers

RBF Network



Basis Functions

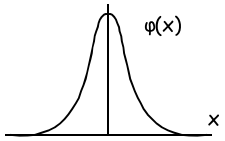
Takes in a scalar number and returns a scalar number which is the output of the hidden layer node.

Examples of common basis functions:

Thin Plate Spline: $\phi(x) = x^2 \log x$

Gaussian: $\phi(x) = \exp(-x^2/2w)$

where w represents its width.



What happens in a hidden layer node?

Euclidean distance between input and centre vector found.

n dimensional Pythagoras $\sqrt{((x_1-c_1)^2 + (x_2-c_2)^2 + .. (x_n-c_n)^2)}$

Distance produced is passed through the basis function to produce the output of the hidden layer node.

Specific Equations

Let there be m 'neurons' in hidden layer and k outputs

Let the input be $\mathbf{X} = [x_1, x_2 \ldots x_n]$

Let the rth centre be $\boldsymbol{C}_r = [c_{r1}, c_{r2} \ldots c_{rn}]$

Let the basis function be $\phi(x)$

Let the 'weight' of connection from r^{th} neuron to j^{th} output be w_{rj}

Then the jth output is:

$$y_j = w_{0j} + \sum_{i=1}^{m} w_{ij} * \phi(|\mathbf{X} - \boldsymbol{C}_i|)$$

where $|\mathbf{X}-\boldsymbol{C}_i| = \sqrt{\sum_{rst1}^{n} (x_r - c_{ir})^2}$

Example RBF to evaluate XOR

The Centres are chosen as: $C_1 = [1\ 1]^T$ and $C_2 = [0\ 0]^T$

The weights of the output node are $w_0 = 2.84$, $w_1 = w_2 = -2.5$

We will use Gaussian basis function $\exp(-\text{dist}^2)$

The input, let X = [0 1]; For XOR the desired output is [+1]

Distance Squared of X from C_1 [1 1]

$(0 - 1)^2 + (1-1)^2 = +1$

Output of neuron 1 = Gaussian (+1) = $e^{-1} = 0.3679$

Distance Squared of X from C_2 [0 0]

$(0 - 0)^2 + (1 -0)^2 = +1$

Output of neuron 2 = Gaussian (+1) = $e^{-1} = 0.3679$

Output = (0.3679*-2.5) + (0.3679*-2.5) + (2.84) = 1.0006

In MATLAB

```
function [output, houts] = rbf_calc (rbfnet, input)
% [OUTPUT, HOUTS] = RBF_CALC (RBFNET, INPUT)
% RNFNET is a struct with three fields
%   CENTRES each row of matrix are centres for one neuron
%   RADIUS radius^2 (of Gaussian basis function) of each neuron
%   WEIGHTS the weights (including bias) for OUTPUT
% Prof Richard Mitchell 31/07/03
houts = [1];     % outputs of hidden neurons: first is 'input' to bias = 1
for ct=1:size(rbfnet.centres,2)                  % for each hidden node
   distsq = sum((input'-rbfnet.centres(:,ct)).^2);       % dist squared
   houts = [houts, exp(-distsq/(2*rbfnet.radius(ct)))];
end                                              % add to houts, output of basis func
output = dot (houts, rbfnet.weights);       % compute actual output
```


Continued and Then…

```
>>rbfn = struct('centres', [1, 1; 0 0]', 'radius', [0.5 0.5],
                'weights', [2.84 -2.50 -2.50])
>> rbf_calc(rbfn, [0 1])
ans =
   1.0006
>> xor=[0 0; 0 1; 1 0; 1 1];                    % xor test set
>> for ct = 1:4, xo(ct) = rbf_calc(rbfn, xor(ct,:)); end; xo
xo =
   0.0017    1.0006    1.0006    0.0017
```

So can solve XOR (and hence other 'hard' problems)

But – how are centres chosen, radius found and weights set?

Designing an RBF

How to choose centres

Centres chosen to represent input training data set.

Correct choice of centres is critical for good performance.

The number of centres

Too many or too few leads to inaccuracy.

Now, no rigorous formal method to find optimum number.

Methods for choosing centres:

Simple distribution

Uniform distribution over data space

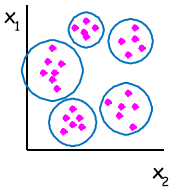
Gaussian distribution over data space

Distribution related to the data distribution

Eg. Use of clustering algorithms

Cluster

Group together data points in n-dimensional space.
Easy in 2D …
If know have k clusters, and have initial guess of where are, use K-means (McQueen, 1967)



REPEAT

allocate each point to nearest centre : use Euclidean distance

Centres := Mean (points in cluster)

UNTIL centres don't move

QError := Mean (distances from centre)

Centres move to where data dense.

If don't know k. do above with K = 1… til Qerror just better

Finding 'Radius' of Each Centre

Once each centre is found, need to find the radius of the Gaussian basis function used on that centre

Radii should be set so the Gaussian from one centre overlaps with near centres to a certain extent

This ensures 'smooth' transition across the data space.

Picton: for each centre find the P (typically 2) closest centres

If C_j is centre of interest and $C_{j1}..C_{jp}$ are closest centres to C_j

Then the radius is set by: $\sigma_j = \sqrt{\frac{1}{P}\sum_{i=1}^{P}\left(\left|C_j - Cjp\right|\right)}$

Remember Gaussian is $\phi_j(x) = \exp\left(-\frac{x^2}{2\sigma_j^2}\right)$

Training the RBF

Once the centres and their radii are chosen, all that is left is to choose the weights.

This is (compared with mlps) very easy

As for mlps, have training set with inputs & expected outputs

For pth item in training set calc output of each hidden node

$h_{ip} = \phi(X_p, C_i, R_i)$

X_p is pth input set from training set

C_i and R_i are centre of radius

The network output (whose value we know) is

$$y_p = w_0 + \sum_i w_i * \phi(X_p, C_i, R_i) = w_0 + \sum_i w_i * h_{ip}$$

Thus, For The Training Set We Can Say

$$y_1 = w_0 + w_1*h_{11} + \ldots w_n*h_{n1}$$
$$y_2 = w_0 + w_1*h_{12} + \ldots w_n*h_{n2}$$
$$\vdots \qquad \vdots$$
$$y_m = w_0 + w_1*h_{1m} + \ldots w_n*h_{nm}$$

$$Y = \begin{bmatrix} y_1 \\ y2 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} 1 & h_{11} & .. & h_{n1} \\ 1 & h_{12} & .. & h_{n2} \\ 1 & \vdots & & \\ 1 & h_{1m} & .. & h_{nm} \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \\ w_n \end{bmatrix}$$

More compactly : **Y = H W** - we want **W** & know **Y** and **H**

If number of items in training set equal number of weights, there is an exact solution, if equations are independent

Usually more items in the training set, solve with min. error^2

If H has all the H values (including dummy 1s), and Y the expected Y values, in MATLAB, W is found simply by

```
>> W = H \ Y          % means matrix div of H into Y
```


MATLAB Session

Using the network given earlier, and the XOR test set

```
>> for ct = 1:4, [a,h(ct,:)] = rbf_calc(rbfn, xor(ct,:)); end,  h
h =
   1.0000   0.1353   1.0000
   1.0000   0.3679   0.3679
   1.0000   0.3679   0.3679
   1.0000   1.0000   0.1353
>> rbfn.weights = h\[0;1;1;0]          % to find weights
ans =
   2.8413
  -2.5027
  -2.5027      % more accurate than values quoted earlier
```


Example: Demand = f(Temp, Illum)

This is same problem as used for MLPs ....

Have Training Set and Unseen Set – cluster training data

For simplicity, radii set of circles which encircle all of cluster

Set RBF network based on cluster centres, with zero weights

Compute the output of the basis functions

Use these and training values of Demand to compute weights

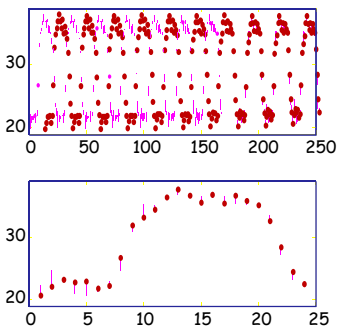
Find Av. Mean Square Error (MSE) of Demand and RBF o/p

Input Unseen Data to trained RBF and compute MSE

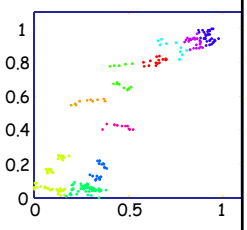
Training error is 1.18, Unseen error 1.55 – not bad

Better than MLP, though more work needed on MLP.

Errors on Trained and Unseen Data



Also clusters



Comparison with RBFs and MLPs

Differences:

Feature	MLP	RBF
Variables:	Weights and Thresholds	Centres and Coefficients
Speed of Training:	Slow Comp. expensive	Fast(er) Less computation
Accuracy:	Very good	Not as good
Math Description:	Difficult	Easy

Similarities:

Learn from a set of training data.

Ability to generalise from training data.

Store information in a distributed manner.

Can be implemented using parallel processing.

9 : Weightless Neural Networks

The standard MLP type network has various drawbacks, one of which is the time it takes to learn.

An alternative type of network, almost unique to the UK, is the Weightless Neural Network – these are also called

n-tuple networks or RAM based networks.

These have a very different model of a neuron – a memory

These neurons have no weights – hence 'weightless' nets

Learning is also different and much simpler

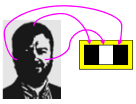
Being based on memories, are implementable in hardware – WISARD was first commercial neural network system

We will investigate the standard system and generalise

The n-tuple Neuron

Weightless networks arose from Bledsoe & Browning's work (1959) on n-tuples (n bits sampled from binary input)

pre-processed to

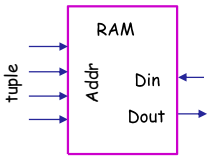


n-tuple neuron is standard RAM

A n-tuple is put on the input address lines of the RAM.

To learn, a value is written into the specified address.

To analyse, read from the addressed location.



Simple (Impractical) Use

Suppose (as example) we want to recognise images of faces

Initially, clear all locations in RAM

Get a binary image: learn it by connecting each pixel to RAM input, write a '1' at addressed location.

Other images could be learnt also.

When present a new image, if addressed location has a '1', image recognised otherwise, the system not learnt it.

But for 256*256 binary image, RAM needs 2^{16}=65536 inputs and have 2^{65536} locations: MATLAB says is infinite!

System only says yes/no : learnt or not learnt an image

It learns what it is taught, but cannot interpolate

it cannot generalise.

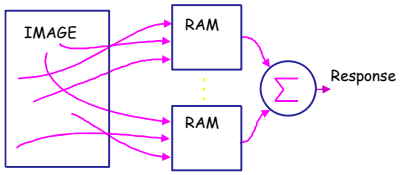
Practical Configuration

Have not one but many smaller RAM neurons.

Each RAM is connected to only some of the image

It is responsible for learning/analysing part of the image

Normally each bit in image is an input to 1 RAM only – but it is possible to 'oversample' to improve classification.



Analyse: count how many RAM neurons 'fire' (output 1)

Class Discriminators

A group of RAM Neurons is called a Class Discriminator

Typically many (similar) examples of one pattern class (eg various images on one person's face) are taught

Then the Discriminator is able to recognise

an image it has been taught AND

one similar to, but not identical to one already taught.

e.g. RAM 1 might recognise a tuple from image 5

RAM 2 might recognise a tuple from image 8, etc.

In practice an image is 'recognised' if number of RAMs which output 1 exceeds a threshold (say 90% - but depends on amount of noise, and accuracy needed).

Multiple Discriminators

Note can 'teach' different classes in one discriminator

e.g. images of two faces can be taught

Then system can say if it recognises an input

but it will not be able to say which face it is

To discriminate between classes,

Have one discriminator for each class

Teach each class into its own discriminator ONLY

When analyse, count firings of ALL discriminators

Image belongs to discriminator with most 'fires'

Memory Requirements

In a Discriminator employing:

Input vector of size ***R***.

k-times over-sampling.

Using a tuple size of ***n***.

Memory of each RAM is defined as Z, where $\mathbf{Z} = 2^n$

M RAMs are needed.

Where $\mathbf{M} = k \times R / n$.

The memory requirement of the entire network is thus :

MEMORY = M (rams) x Z (bits per ram)

e.g. 256*256 image, 8 bit tuples, no over-sampling (***k*** = 1)

M = 1 * 256 * 256 / 8 = **8192** and **Z** = 2^8 = **256**

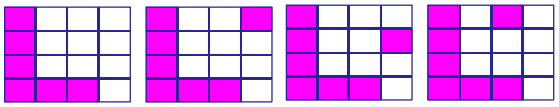
MEMORY = 8192 * 256 = 2097152 bits

Generalisation in a n-Tuple Network

How many input patterns will result in maximum output of four RAMs firing?

1	2	1	3
4	3	2	2
2	1	3	3
4	4	4	1

Tuple Maps for 4 RAMS Used on Images 1..4

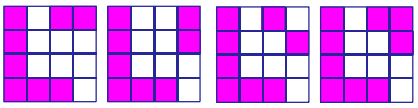


Tuples learnt from each image

	Tuple1	Tuple2	Tuple3	Tuple4
Image 1	1000	0001	0000	1111
Image 2	1000	0001	1000	1111
Image 3	1000	0011	0000	1111
Image 4	1100	0001	0000	1111

Hence consider these images (5..8)

These are [2+3], [2+4], [3+4], [2+3+4]



	Tuple1	Tuple2	Tuple3	Tuple4
Image 5	1100	0001	1000	1111

Each tuple has been learnt, so system will recognise image 5.

Exercise – verify system also recognise images 6..8.

So training set of 4, generalisation set of 8.

Choice of Tuple Size

Small tuple size → small memory.

e.g. Tuple size 4 Memory size 16

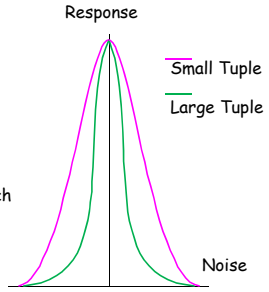
Tuple size 3 Memory size 8

Smaller memory, fewer different tuples can be learnt before the memory filled (neuron saturates)

Then can't discriminate between patterns : o/p always 1

When using a small tuple size, few patterns should be learnt into each Class Discriminator.

If larger tuple size network more sensitive to noise.



Implementation Issues

In principle, could have parallel machine

Each RAM learns/analyses at the same time

For software, and some hardware implementations

Process each RAM in turn; treat all RAMs as one memory

bool array [1..NumRams, 1..2upN]

Tuple Mapping –where in Image should a RAM sample?

NB – each RAM always samples from same locations

Could sequentially map eg RAM n use locations n*8-7..n*8

Can give poor discrimination of vertical/horizontal bars

Better to use random maps (but always in same order)

Usually sample each location in image once only

Algorithm

First, set all locations in each RAM to 0

Learn (one image)

For all RAMS

Sample n-bits (to form a tuple)

Write '1' in location with address tuple in given RAM

Analyse (one image)

NumFire := 0;

For all RAMs

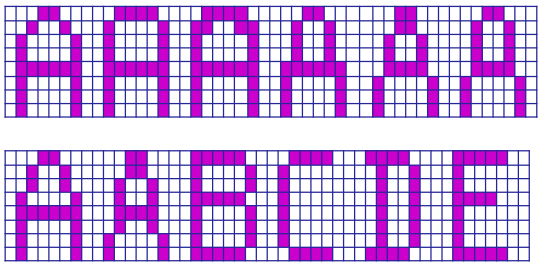
Sample n-bits (to form a tuple)

If location with address tuple in RAM = 1, INC(NumFire)

IF NumFire > Threshold, Image is Recognised

Let us show some experiments on character data

What the characters look like



8 A's a B, C, D and E Train on some A's, test on rest

Experiments – using MATLAB code

```
>>[chars, tmap] = www_makedata;  // make data+tuple map
>> discrim = wnn_simple (chars(1), tmap, 4);
>> wnn_simple (chars([1:12]), tmap, 4, discrim)
100  69  81  63  44  50  69  31  25  25  25  25
% Taught 1, recognise it, not good at others
>> discrim = wnn_simple (chars([1:4]), tmap, 4);
>> wnn_simple (chars([1:12]), tmap, 4, discrim)
100 100 100 100  81  88  88  56  38  31  31  31
% Taught 1..4, recog 1..4, ok at 5..7, not recog BCDE
>> discrim = wnn_simple (chars([1:5, 8]), tmap, 4);
>> wnn_simple (chars([1:12]), tmap, 4, discrim)
100 100 100 100 100  94  88 100  38  31  38  31
% Taught 1..5,8; recog all 'A's and not BCDE - good
```

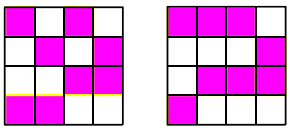

Now Raise Tuple Size to 8

```
>> discrim = wnn_simple (chars(1), tmap, 8);
>> wnn_simple (chars([1:12]), tmap, 8, discrim)
100  50  63  38  13  25  50  13   0   0   0   0
% still only recog one taught
>> discrim = wnn_simple (chars(1:4), tmap, 8);
>> wnn_simple (chars([1:12]), tmap, 8, discrim)
100 100 100 100  50  75  75  38  13  13  13  13
% Recog taught, not so good on other A's as before
>> discrim = wnn_simple (chars([1:5,8]), tmap, 8);
>> wnn_simple (chars([1:12]), tmap, 8, discrim)
100 100 100 100 100  88  75 100  13  13  13  13
% Not so sure non taught A's – better rejection of BCDE
```


What If Use Linear Mapping?

```
>> discrim = wnn_simple (chars(1), [1:64], 4);
>> wnn_simple (chars([1:12]), [1:64], 4, discrim)
100  75  75  75  50  63  75  38  31  38  50  31
% Poorer at discrimination
>> discrim = wnn_simple (chars([1:4]), [1:64], 4);
>> wnn_simple (chars([1:12]), [1:64], 4, discrim)
100 100 100 100  63  88  75  50  31  63  50  38
% Indeed poorer at discrimination
>> discrim = wnn_simple (chars([1:5,8]), [1:64], 4);
>> wnn_simple (chars([1:12]), [1:64], 4, discrim)
100 100 100 100 100 100  75 100  31  63  50  38
% C is thought to be quite like A!
```


Tuple Maps to Optimise Discrimination



From Bishop, Crowe, Minchinton & Mitchell, 1990 (IEE colloq)
Evolutionary Learning to Optimise Mapping in n-tuple networks

e.g. Two 16 bit patterns with four bits different. tsize = 4.

If tuples sampled by columns, 3 of 4 tuples same, so if taught first image, second image is 75% like the first.

If tuples sampled by rows, all tuples different.

So the way the tuples are sampled can be significant.

Optimisation for Max Discrimination

Select, at random, some mapping
Get measure of discrimination
while (insufficient discrimination)
  Select other mapping by mutation of bits in mapping
  Get measure of discrimination
  if (better) adopt this mapping and its measure

Note amount of mutation decays (as in simulated annealing)

Measure of discrimination done by learning class A and class B, analysing both and looking at number of fires.

On characters – much better discrim between c & e; i & l.

Next week

Look at other improvements/variations of WNNs

10. More Weightless Neural Networks

The basic RAM Neuron based Weightless Neural Network has been described.

This comprises a discriminator – an array of RAM neurons

One discriminator used for each class of data to be learnt if to able to discriminate (ie distinguish between classes)

This week some advances / alternatives will be discussed

Handling non binary data …

thermometer codes, CMAC, Minchinton cells

Associative Networks and Pattern Separation

(Briefly) Alternative Configurations

Stochastic Diffusion Search and Weightless Nets

Handling Non-Binary Input Data

What if images comprise grey levels not just black white

Want to be able to cope with small changes in lighting

Could turn grey level into series of 0s and 1s (say 8 bits)

Then data to process is 256*256*8 bits

But – need more RAMs

a change of grey from 3 to 4 involves 3 bit changes

thus generalisation will be poor

One solution is to use gray code

0 to 7 is: 000 001 011 010 110 111 101 100

So changing 3 to 4 is 010 to 110 just one bit change

But too many RAMs needed still

Threshold and Thermometer Coding

Simple solution – choose a **Threshold**

if gray value ≥ threshold, tuple bit = 1 else tuple bit = 0

Image now in effect 256*256*1 so same num of RAMs

But if lighting changes a little and many values near threshold there can be quite a large tuple change.

More advanced – multiple thresholds –**Thermometer Code**

Replace (say) 0..255 by (say) 5 patterns

v < 50	v < 100	v<150	v<200	rest
0000	0001	0011	0111	1111

Image now 256*256*5, so need 5 times as many RAMs

But system less susceptible to lighting changes

CMAC Coding + Gray Code

Each value in Input, I[x] is processed to produce K values P[1..K]:

P[j] = Gray ((I[x] + K - 1 - j) / K)

where Gray (v) returns the Gray coded version of integer v.

If P[j] is r-bits, each value I[x] mapped to K*r bits of data.

NB (I[x] + K - 1 - j) / K produces some values v, some v+1;

These are bits which are sampled so as form the tuples.

Idea, if d = abs ($I[x_1]$ - $I[x_2]$) < K, then Hamming distance between encoded versions of these values incr's with d.

Thus network can cope with small changes in lighting.

But, like Thermometer coding, needs much more memory:

k * r * InputSize / n RAMs needed

Minchinton Cells

These are simple processing elements placed between input data and the tuple forming elements

Consider as being between input & RAM address inputs

Let I(x) be value at position x in input data I

Simplest cell I(x) > constant This is thresholding

Type 1 cell $I(x_1) > I(x_2)$ Compares two random points

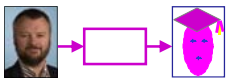
If lighting changes, for much of image, grey value change by constant amount, so difference unchanged. So type 1 cell makes system more tolerant of lighting changes.

Does not increase number of RAMs. Seems best method.

S Lauria, R.J.Mitchell: "*Pre-Processing Grey Level Data for Weightless Neural Networks*", Proc CESA '98, 671-675

Weightless Associative Networks

Want System to do



RAMs map input to output.

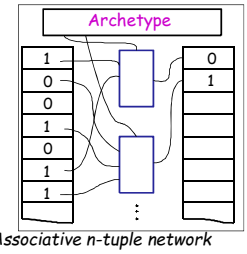
System taught 'Archetype'

Archetype and Output same size

Over-sample by tuple size if want Output same size as Input.

When learning into r^{th} RAM, store the value of r^{th} pixel in 'archetype' image.

When analyse, values output from RAMs should be archetype (or close to it).



Associative n-tuple network

Pattern Separation – Binary Images

D Aitken J.M.Bishop R.J.Mitchell S Pepper : *Pattern Separation in Digital Learning Nets*", Elec Lett, 25:11, pp: 685-686 (1989)

For storing archetypes of many classes in a discriminator

But one archetype may want 1 in a location, another a 0.

So define RAMs to have four states

- GROUND – not learnt
- State 0 – equivalent to a '0'
- State 1 – equivalent to a '1'
- CLASH – where tried to override a '1' with a '0'.

Initially all RAMs are in GROUND state.

Here want Output of same size as Input, so oversample

Learning and Analysis

Learning - for storing value at address 'tuple' in r^{th} RAM.

```
IF RAM (tuple) = GROUND THEN
    RAM (tuple) = INPUT(r)
ELSEIF RAM (tuple) ~= INPUT(r) THEN
    RAM (tuple) = CLASH
```

Analyse - for getting value from addres tuple in rth RAM

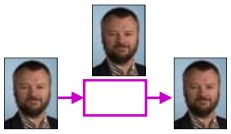
```
IF RAM (tuple) = '0' OR '1'
    THEN value = RAM (tuple)
    ELSE value = INPUT (r)
```

ie if don't know r^{th} value (as RAM does not have r^{th} value of the archetype), best guess is r^{th} value in input.

Auto-associative Network

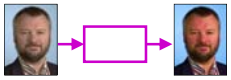
An autoassociative network

Archetype is the 'noise free' version



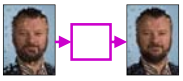
Train by showing different versions of RJM, storing the archetype

So, if then show an image, should get something like archetype

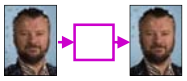


Then Use Feedback

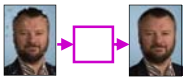
But, the output is, hopefully, closer to the archetype than the input



So now show the output to the MLP - feedback



Output should be even closer … etc



Can do colour/greyscale, by having states 0..n (for colour/grey) as well as GROUND and CLASH, but basically operation same.

Other Types of Weightless Networks

Note, other types exist

- P-RAMs and PLNs - Probabilistic Neurons
  - neuron stores the probability that it will fire.
  - These are used in feedback circuits
- G-RAMs - Generalised Neurons (Igor Aleksander)
- ADAM - a different form of associative network
  - this is the work of Jim Austin at York.
  - it is a two stage network

In addition, have hybrid systems

- So consider WNN + Stochastic Diffusion Search (SDS)
- SDS – started by Mark Bishop (ex Cyb); also Slawek

Stochastic Diffusion Search - SDS

*"**Global** optimal search via stochastic communication in a population of discerning agents."*

Each 'agent' is a potential solution to part of the problem,

- (Contrast, GAs have solutions to whole problem)
- Each can discern whether it has a partial potential solution

Agents communicate their solutions to others

Dynamic cluster of agents stabilises round the best fit

- If noise free data, find solution else x% of agents have best soln

Related in a way to a colony of ants searching for resource

- initially ants embark on random walk
- if ant not found resource but encounters another
  - if other has found resource, ant joins it
  - else ant goes on randomly.

Application – and Basic Algorithm

Suppose look for 'fred' in search space 'bertfredfraneddy'
Solution is 'fred' is at position 4 (strings start from 0).
The features are letters; agents define possible positions.
S is structured set of features defining search space
and T is structured set of features defining the target
Agents are population of independent cells: map T to S
The 'vanilla' SDS algorithm

```
Initialise (Agents)
while (! Terminate(Agents) )      % while not terminated
    Test (Agents, T, S)           % are agents at possible soln
    Diffuse(Agents)               % update their states
```


Initialisation

Each agent is assigned a possible solution (mapping)
In string example, it is a potential position in the string
If no prior knowledge choose randomly (cf ant walk)
But can employ a-priori knowledge to continue search

Termination

Simplest case – stop if the largest number of agents with correct mapping constant over time
In noise – stop if num correct agents > threshold
'Strong halting criteria' (Nasuto & Bishop 1999) guarantees convergence to global optimum position
If 'fred' not exist in S, but 'fre' did, search stabilises with (an almost constant ~75%) number of agents testing ok.

Test Phase – For Each Agent

Each agent has possible position and tests a random selected feature from the target T
Illustrate this with respect to string search example
S = 'bertfredfraneddy'
T = 'fred', a 4 letter string, so a feature is offset 0..3
Assume an agent is at position 7 and feature offset is 2
The test here will be to see whether S[7+2] = T[2]
i.e. S[9] = 'f'; T[2] = 'e' so agent does not match solution
If, however, agent is position 10; S[10+2] = 'e' as is T[2]
so agent is at a possible solution (though not global one)

Diffusion Phase

If an agent has a possible solution,
a) it maintains that possible solution
b) next time it will test another feature
(so if solution not global, next time test may return false)
If an agent does not have a possible solution
It selects at random another agent
if (other agent has a possible solution)
agent takes that solution
else agent selects a new mapping at random
{ compares well with ant search method }

Time Complexity and Alternatives

Convergence Time t_c of SDS (in Bishop & Nasuto 1998)
M = search space size; N = num of agents

$$\text{If } M > \frac{N^{1/N-1}}{N^{1/N-1}-1}\ t_c = O\left(\frac{M}{N}\right) \text{ else } O\left(\frac{1}{\log N}\right)$$

'Vanilla' SDS extended (Nasuto) to offer different balance re exploration of search space & exploitation of solns
In diffusion phase positive agent randomly selects another
In 'Context Free SDS'
if other agent also active, selector becomes inactive
In 'Context Sensitive SDS'
if other agent active same mapping, selector set inactive

Application : Finding Eyes in Faces

This was our entry into a BT 'competition'
S is image of a face, T is weightless NN trained on eyes
T is thus trained on a series of images of parts of faces.
Each feature of T is RAM Neuron with assoc tuple positions
When testing agent, choose one of the RAM neurons in T
Get tuple from S as normal, by sampling at 'tuple'
positions offset by position of agent
Test to see if associated RAM has '1' at 'address' tuple
Used on gray scale images, so Minchinton cells used.
Worked reasonably well at finding positions
Concept can be extended for rotations, scaling and 3D
Here endeth the course – you complete the assignment