Single Layer 'Perceptron' Networks

We have looked at what artificial neural networks (ANNs) can do, and by looking at their history have seen some of the different types of neural network.

We started looking at single layer networks based on Perceptron or McCulloch Pitts (MCP) type neurons

We tried applying the simple delta rule to the AND problem

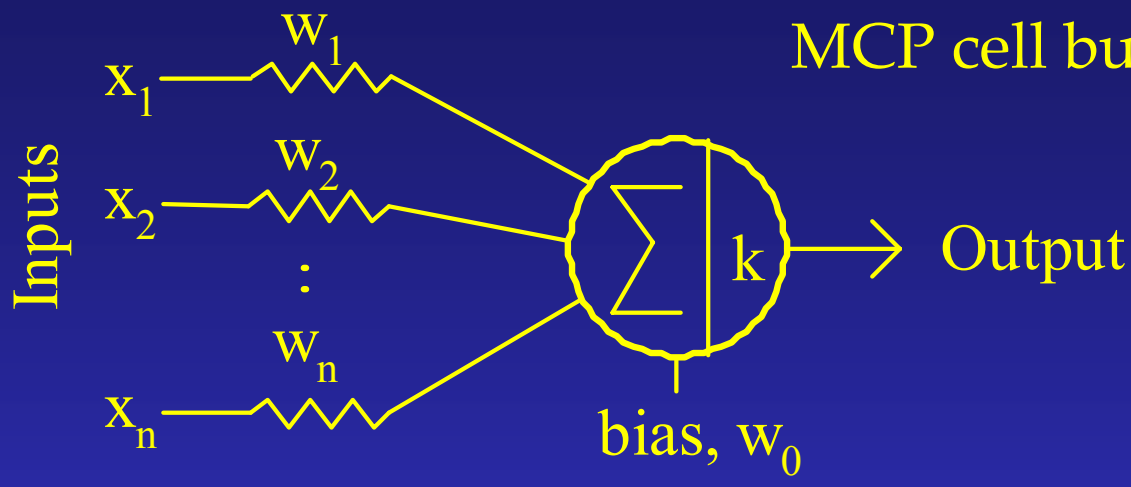
In this lecture we will show some MATLAB functions to allow us to finish learning AND, and then do OR

We will then see that we can't, at this stage, do XOR

This will lead to multi-layer perceptrons.

We will also demonstrate that the Delta Rule does follow the steepest gradient down the weight error space

Reminder Simple Linear Neuron



Output, $O = k * (\Sigma (x_i * w_i))$ NB $x_o = 1$; k is often 1

To train, n input values and corresponding known output

In fact have training set, with many such n+1 value sets

Pass each in turn, calc O, if T is target, change weights by

$\Delta w_i = \eta (T - O) x_i = \eta \delta x_i$ the delta rule

Then pass next item from training set, etc.

MATLAB for Single Layer Networks

As you will be able to use MATLAB in the exam, here is an intro to using MATLAB for single layer networks

Two functions are presented, here is the first

```
function node = sdr_makenode(initweights);
% NODE = SDR_MAKENODE (INITWEIGHTS)
% makes a structure with weights INITWEIGHTS
% an output and a delta field
% Dr Richard Mitchell 25.7.03
node = struct('weights', initweights, 'output', 0, 'delta', 0);
```

Generates a structure with fields for data in neuron.

Call by, for instance:

```
>> node = sdr_makenode([0.05, 0.1, -0.2]);
```

Next MATLAB Function to Learn

```
function [node, sumsqerr] = sdr_learn (node, tset, lrate)
% [NODE, SUMSQERR] = SDR_LEARN (NODE, TSET, LRATE)
% node is struct('weights', [w0..wn], 'output', 0, 'delta', 0);
% applies each row in training set & adjusts weights suitably
% Dr Richard Mitchell 25.7.03
sumsqerr = 0;
for r = 1:size(tset, 1),                          % for all rows in tset
   invec = [1, tset(r, [1:size(tset,2)-1])];      % 1 and input
   node.output = dot(invec, node.weights); % compute output
   node.delta = tset(r,size(tset,2)) - node.output;% error
   sumsqerr = sumsqerr + node.delta^2;      % add to error sum
   node.weights = node.weights + lrate * invec * node.delta;
end                                               % update weights
```

Notes on MATLAB code

tset = [0 0 0; 0 1 0; 1 0 0; 1 1 1] is 3 column 4 row matrix

size(tset,1) is number of rows; size(tset,2) is number of cols

invec = [1, tset(r, [1:size(tset,2)-1])]; % input vector being 1 (for bias) then columns 1 to 2 in row r of tset

dot(invec, node.weights) is the dot product which is in fact invec(1)*node.weights(1) + invec(2)*node.weights(2) + invec(3)*node.weights(3) ie weighted sum inc bias

MATLAB session:

```
>>tset=[0 0 0; 0 1 0; 1 0 0; 1 1 1];      % define training set
>>[node, sse]=sdr_learn(node, tset, 0.1);
>> sse
   1.1676                                 % error after one 'epoch'
```

```
>> for ct=2:20, [node, sse(ct)]=sdr_learn(node, tset, 0.1); end
>> sse                                          % see how error drops
sse =
1.1676    0.8152    0.7096    0.6551    0.6123    0.5741    0.5400
0.5097    0.4831    0.4599    0.4397    0.4221    0.4069    0.3937
0.3822    0.3723    0.3637    0.3562    0.3497    0.3441
>> node.weights                                 % 'final' value of weights
ans =
   -0.1260    0.4589    0.3990
>> for r=1:size(tset,1),                        % compute training set
        node.output = dot([1, tset(r, [1:2])], node.weights);
        [tset(r,:),node.output], end
ans =    0.0000    0.0000    0.0000   -0.1260
ans =    0.0000    1.0000    0.0000    0.2730
ans =    1.0000     0.000    0.0000    0.3329
ans =    1.0000    1.0000    1.0000    0.7319     % as last week
```

Now Do For OR Function

otset = [0 0 0; 0 1 1; 1 0 1; 1 1 1];

Learn 100 times; sse down to 0.3086

node.weights = 0.2769 0.4451 0.4729

If we test the result (show input, target and actual output)

0.0000	0.0000	0.0000	0.2769
0.0000	1.0000	1.0000	0.7498
1.0000	0.0000	1.0000	0.7220
1.0000	1.0000	1.0000	1.1949

If threshold is 0.5 say, have learnt OR function

Note number of epochs needed to learn, for a given learning rate, depends on initial weights (and hence initial error)

Now Do The XOR Function

```
>> etset=[0 0 0; 0 1 1; 1 0 1; 1 1 0];
>> node=sdr_makenode(randn(1,3));
>> for ct=1:100, [node, sse(ct)]=sdr_learn(node, etset, 0.1);
   end
>> sse(100) = 1.2345
```

If we test the result (show input, target and actual output)

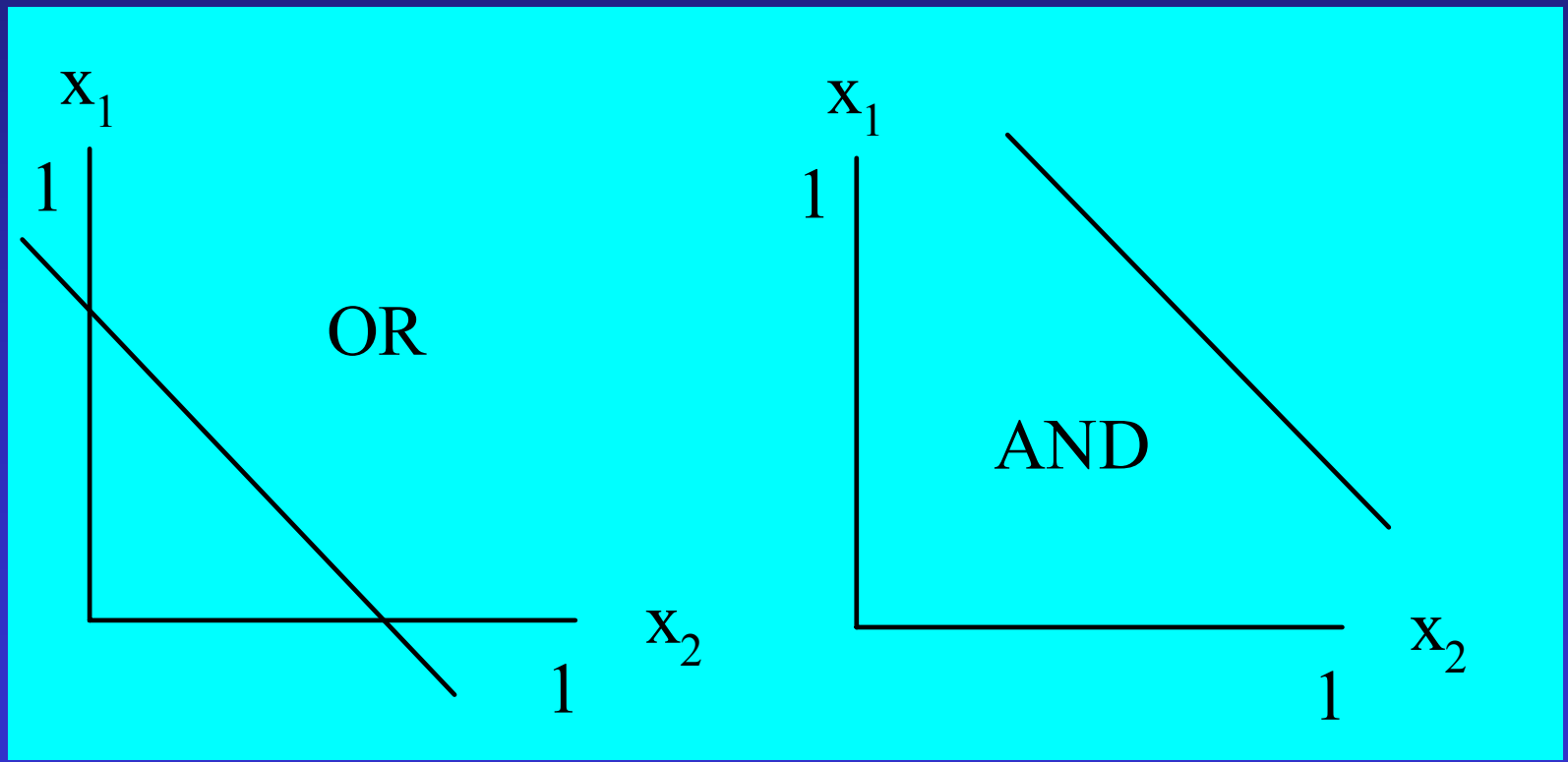
0.0000	0.0000	0.0000	0.5544
0.0000	1.0000	1.0000	0.4997
1.0000	0.0000	1.0000	0.4441
1.0000	1.0000	0.0000	0.3894

Clearly we have failed to learn the XOR problem

Linear Separable Problems

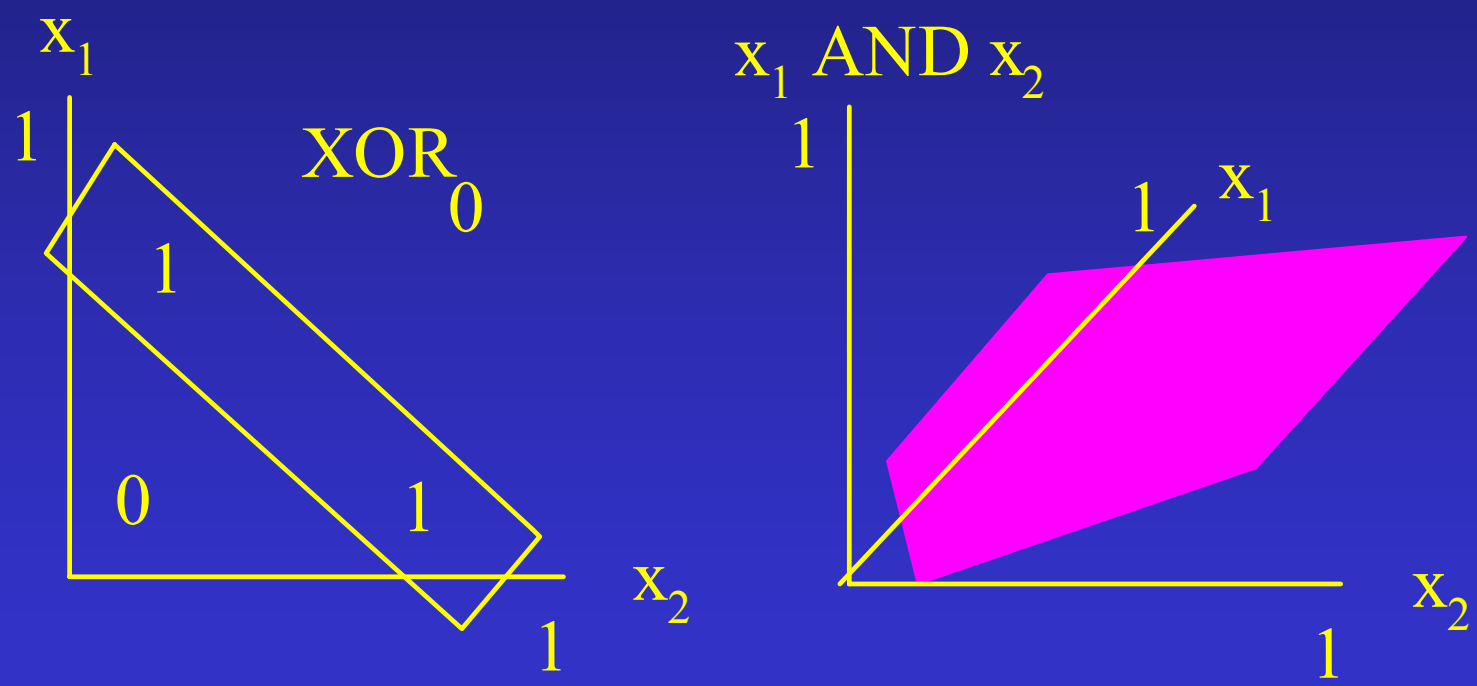
A two input MCP cell can classify any function that can be separated by a straight dividing line in input space

These are 'linearly separable problems'.



XOR Is Not Linearly Separable

A straight line wont separate classes for XOR
If add extra dimension, x_1 AND x_2, linear plane will separate



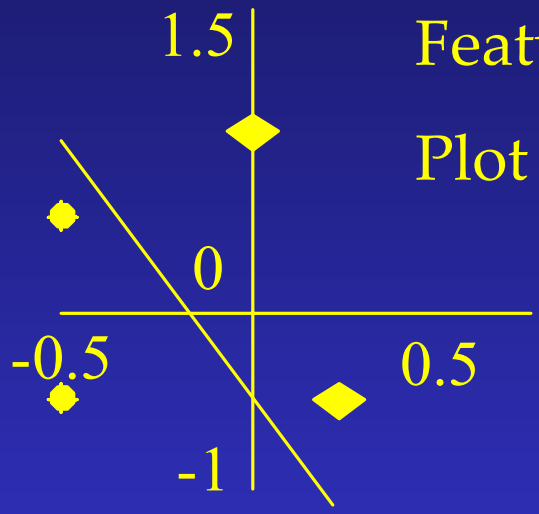
In MATLAB It Works!

```
>> node = sdr_makenode(randn(1,4));
>> etset2 = [0 0 0 0; 0 1 0 1; 1 0 0 1; 1 1 1 0];   % incl x1 AND x2
>> for ct=1:100, [node, sse(ct)]=sdr_learn(node, etset2, 0.1); end
>> sse(100)
ans =
    0.0476
>> for r=1:size(tset,1),
        node.output = dot([1, etset2(r, [1:3])], node.weights);
        [etset2(r,:),node.output], end
0.0000    0.0000    0.0000    0.0000    0.1578
0.0000    1.0000    0.0000    1.0000    0.9218
1.0000    0.0000    0.0000    1.0000    0.9107
1.0000    1.0000    1.0000    0.0000    0.0346
```

On the Separating Line

Consider example with data defined as follows

x_1	x_2	T
-0.5	-0.5	0
-0.5	+0.5	0
+0.3	-0.5	1
+0.0	+1.0	1



Feature Space:

Plot x_2 v x_1

Line separating classes defined by $w_0 + w_1*x_1 + w_2*x_2 = 0$

Line through -0.5,1 and 0,-0.5: so given by $0.5 + 3*x_1 + x_2 = 0$

For points to the right of the line, $0.5 + 3*x_1 + x_2 > 0$

e.g. 0.5 + 3 * 0.3 + -0.5 = 0.9, so Thresh (0.5 + 3 * 0.3 + -0.5) = 1

Cybernetics

Why Delta Rule Does Gradient Descent

For pth item in set, we first calculate the actual output, O_p

$O_p = \Sigma (x_{ip} * w_i)$ NB $x_0 = 1$

Next we calculate the error or delta

$\delta_p = T_p - O_p$

Then, each weight is to be changed by

$\Delta w_i = \eta \delta_p x_{ip}$

η (eta) is the learning rate

We need to define errors, and sum of square of errors used

$E_p = (T_p - O_p)^2$ Over all training set $E = \Sigma E_p$

Note, if there are j outputs $E_p = 1/2 \Sigma (T_{pj} - O_{pj})^2$ where, for instance, T_{pj} is the target for output node j, for pattern p

Proof That It Performs Gradient Descent

To show the Simple Delta Rule performs gradient descent, we must show that the *derivative of the error measure* with respect to each weight is proportional to the weight change dictated by the Simple Delta Rule. i.e.

$$\frac{\partial E_p}{\partial w_i} = k\delta_p x_{ip}$$ which is proportional to Δw_i in delta rule

Using the chain rule

$$\frac{\partial E_p}{\partial w_i} = \frac{\partial E_p}{\partial O_p}\frac{\partial O_p}{\partial w_i}$$

But, $E_p = (T_p - O_p)^2$

So $$\frac{\partial E_p}{\partial O_p} = 2(T_p - O_p) = k\delta_p$$

Continued

For linear neurons, $O_p = \sum_i w_i * x_{ip}$

(x_{ip} is input i for test pattern p, and $x_{0p} = 1$ for bias weight)

if $O_p = w_0 x_{0p} + w_1 x_{1p} + w_2 x_{2p}$, for instance

$$\frac{\partial O_p}{\partial w_2} = \frac{\partial w_0 x_{0p}}{\partial w_2} + \frac{\partial w_1 x_{1p}}{\partial w_2} + \frac{\partial w_2 x_{2p}}{\partial w_2} = 0 + 0 + x_{2p}$$

So, for all i, $\frac{\partial O_p}{\partial w_i} = x_{ip}$

Thus $\frac{\partial E_p}{\partial w_i} = \frac{\partial E_p}{\partial O_p}\frac{\partial O_p}{\partial w_i} = k\delta_i x_{ip}$

So delta rule is prop to grad in Error space

So and but

Over the whole training set, $\frac{\partial E}{\partial w_i} = \sum_p \frac{\partial E_p}{\partial w_i}$

So the net change in w_i after one complete training cycle (one epoch) is proportional to this derivative & hence the Delta Rule does perform gradient descent in *Weight-Error Space*.

NB. If (say for computational reasons), weights are updated after each pattern presentation this will depart from pure gradient descent.

However if the learning rate, η, is small the departure is negligible and this version of the delta rule still implements a very close approximation to true gradient descent.

But what if use sigmoidal activation ?

Delta Rule and Activation Functions

In fact the delta rule needs slight clarification

delta term = 'error' * 'derivative of activation function'

So if z is weighted sum of inputs ,for 'linear activation', O = z

$$\frac{dO}{dz} = \frac{d}{dz}(z) = 1 \qquad \text{So } \delta = \text{error} * 1 = \text{error}$$

But if the neuron had sigmoidal activation $O = \frac{1}{1+e^{-z}}$

$$\frac{d\,O}{d\,z} = (1+e^{-z})^{-2} * -1 * e^{-z} * -1 = (1+e^{-z})^{-2} * e^{-z}$$

$$= O^2 * (1+e^{-z} - 1) = O^2 * (O^{-1} - 1) = O * (1 - O)$$

So δ = error *Output *(1 - Output) {as quoted last week}

Summary, Hard Problems and ….

A single layer network can learn some problems, but not XOR

XOR, like PARITY, Minsky & Papert called Hard

Although 'hard', can solve using algorithmic methods, or:

For any two class k-input problem which is non linearly separable, it is possible to solve using n 'inputs', where n>k, if a suitable 'hyperplane' exists to make problem separable

Or, instead of one layer, have many layers – but that requires there to be an extension of the delta rule.

The discovery and publication of such a rule revived ANNs.

For more on linear separability see Part 1 Lab Expt 2

http://www.cyber.reading.ac.uk/current_students/part1labs/p1expt2.pdf