Week 5 Lecture 9 & 10
Computational Science

From Data to Lines to boundaries – learning to create that boundary

Modelling

a. Relating x to y

$$\text{e.g. } y = m_0 \sum_{i=1}^{n} m_i x_i$$

b. Relating y to time t, and some inputs (we don't take time 't' explicitly – but consider sample(s) e.g

$$y(k+1) = \sum_{i=1}^{m} b_i y(k-i) + \sum_{j=1}^{n} a_j U(k-j)$$

A more generic form of this for training a network or intelligent agent to mimic the system (e.g robot) takes the form

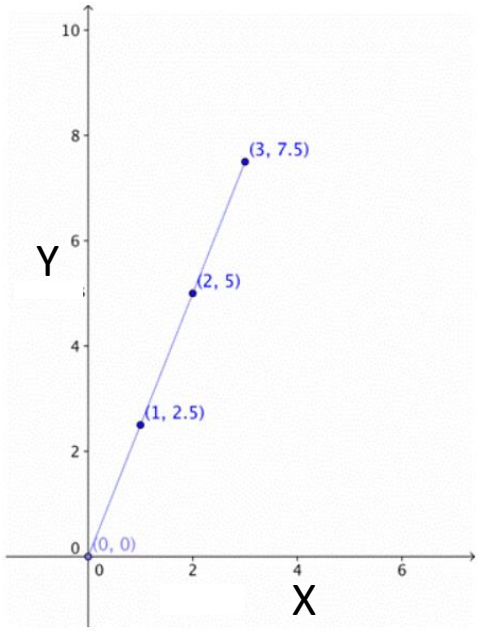
$$y(k+1) = f(y(k), u(k), \theta|k)$$

where θ are the parameters – linear (as in above) or non-linear as in neural networks

c. Recognising patterns, classification

Modelling: Relating x to y

$$y = m_0 \sum_{i=1}^{n} m_i x_i$$



For a straight line – relating Y & X you need two points (data)

Y= mX + C → 2 parameters m, C

If the line is in many dimensions – you will need to estimate n+1 parameters (N is the dimension)
So you will need n+1 data points ← remember this – this n+1 factor repeats itself in different forms for all learning algorithms/paradigms etc

The same holds for fitting curves, non-linear functions etc….the number of points you need is dependent on the number of parameters (weights)

given a set of points $\{(y_1, X_1), (y_2, X_2)\ldots.. (y_m, X_m)\}$ and $X_i = \{x_1, x_2, x_3,\ldots., x_n,\}$ ← n dimensions

Given the previous slide $m > n$

Let $e_1 = y_1 - f(X_1, \theta);\ \theta\ are\ the\ parameters$. Then given the m datapoints; we have n errors $e_1, e_2, e_3,\ldots, e_m$

These are called residuals or errors

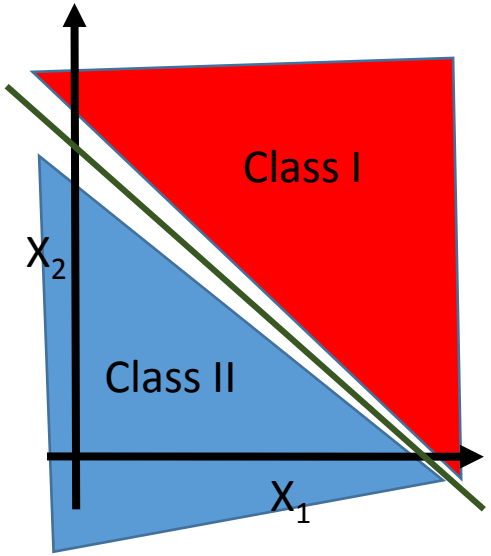
The sum of square of residuals over the n-data points is given by $E^2 = \sum_{i=1}^{m} e_i^2 = \sum_{i=1}^{m} (y_i - f(X_i, \theta))^2$

$$\theta = [X^T X]^{-1} X^T y$$ ⟸ **Least Squares Solution**

$T\ is\ transposing\ row\ vector\ becomes\ a\ column\ vector\ (and\ vice\ versa); rows\ of\ a\ matrixbecome\ columns$

Inverse must exist – hence the m points must be distinct, otherwi it implies dependencies – recall using correlation in datamining

Pattern recognition or Classification



The idea is to determine a line (in this case the green line) which separates the classes

Problem is not simply one of finding a line which fits the data

But one where

- A point on the line results in a zero class
- A point above it belongs to Class I
- A point below it belongs to Class II

The line is called – a decision boundary, separation line

There are an infinite number of lines which can fit the data, this is reduced by trying todetermine the decision boundary.

Again for any given decision boundary – there are infinite other possibilities

What you get is dependent on a number of factors.

$$y = \begin{cases} f(x,\theta) > 0; y = Class\ I \\ f(x,\theta) < 0; y = Class\ II \\ f(x,\theta) = 0 \end{cases}$$

Generally we ignore the third condition
And simply determine $f(x,\theta) = \theta_1 x_1 + \theta_2 x_2 + \theta_0$
In n-Dimensions $f(x,\theta) = \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n + \theta_0$

Example 1

- If temperature is High AND BP is High then call Doc
- If Gun in Hand AND AIM then Fire

The AND Operator

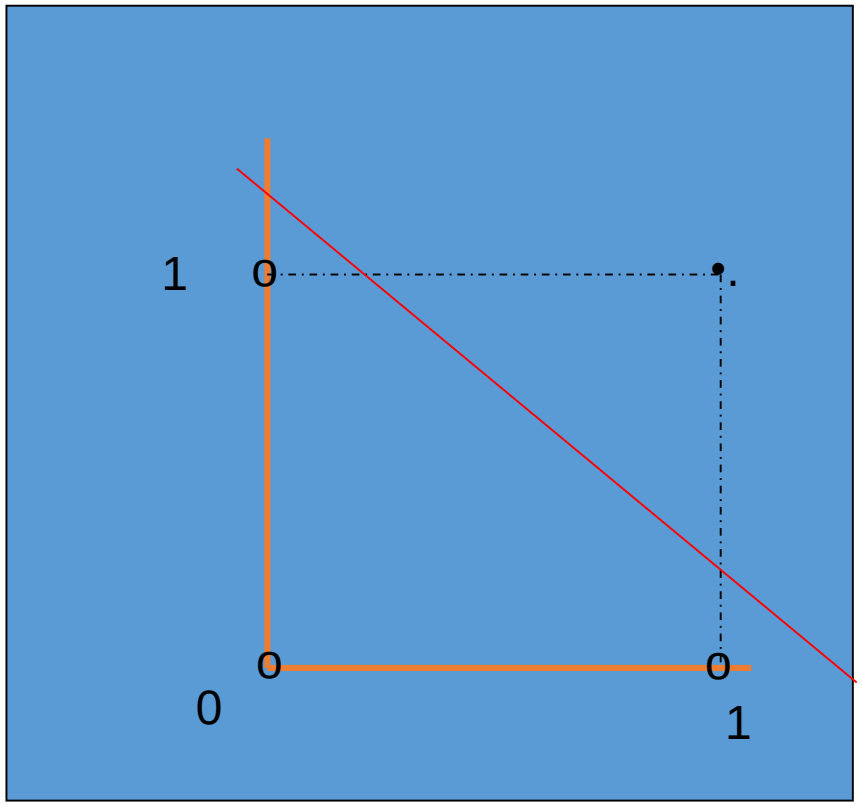
GUN	AIM	Decision
1	0	0
1	1	1
0	1	0
0	0	0

Gun in Hand = yes=1
Gun not in hand = no= 0

Aimed = yes = 1
Aimed = no = 0

Decision = 0 = do not fire
Decision = 1= fire

Example I (Contd)



This requires a straight line to separate the decision regions

The generic form of this Decision Boundary is

$$L : W_1(GUN) + W_2(AIM) + W_3 = 0$$

$$L > 0 \Rightarrow Decision = 1$$

$$L < 0 \Rightarrow Decision = 0$$

Linearly Separable Problem

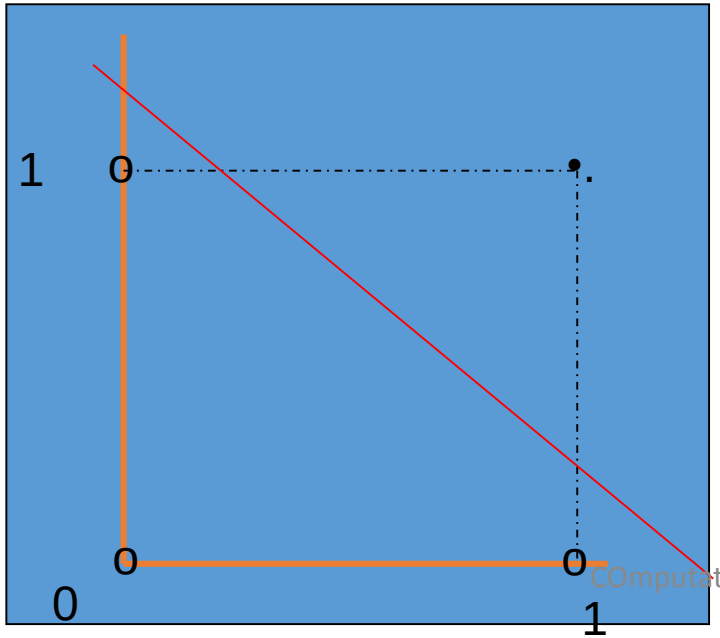
Note: this line is not Unique. There are an infinite number of possible lines.

Example II

- *Pick AK47 OR Colt xxx Then Aim*
- *Switch1 On or Switch2 On then Bulb_On*

X_1	X_2	Decision
1	0	1
1	1	1
0	1	1
0	0	0

- =0 and O=1 in figure below



This requires a straight line to separate the decision regions

The generic form of this line is

$$W_1(X_1)+W_2(X_2)+W_3=0$$

$$L<0 \Rightarrow Decision=1$$

$$L>0 \Rightarrow Decision=0$$

Linearly Separable Problem

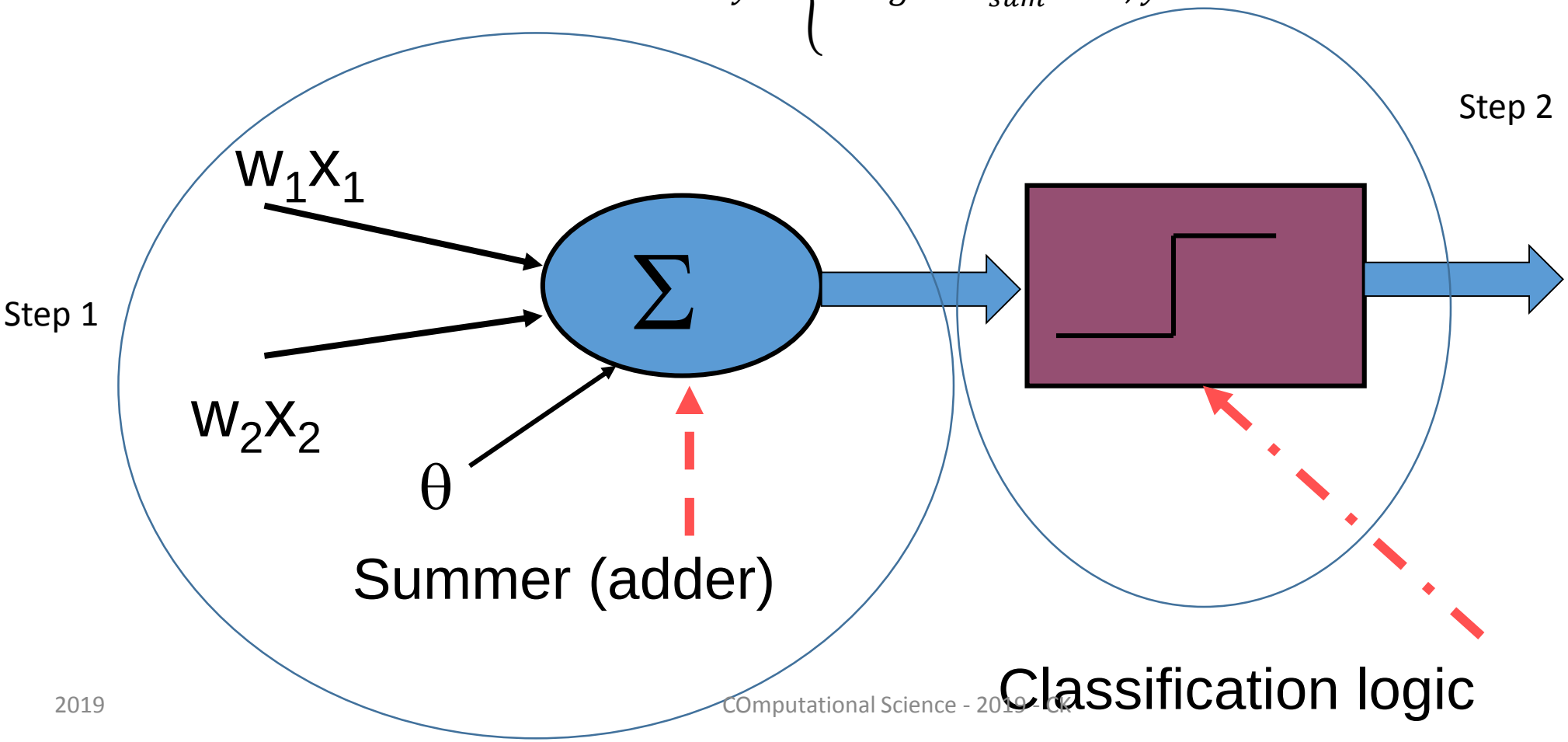
Deconstruct the method

variables x_1 and x2 are weighted by θ_1, and θ_2

Step 1 : Weighted Sum of Inputs;

$Weighted_{sum} = \theta_1 x_1 + \theta_2 x_2 + \theta_0$ ← the bias term (theta_0) has been added –it does not do much

Step 2: Use Weighted Sum to make the decision $y = \begin{cases} Weighted_{sum} > 0; y = 1 \\ Weighted_{sum} < 0 ; y = 0 \end{cases}$



What is needed before we start the training

Input-output pairs → essentially one output many inputs

Definition of a cost function:

This is the error function. Often an error squared function (quadratic cost function)

Error is defined as the difference between the target (desired) output and the output of the perceptron.

Thus if O=output of perceptron and T=target

$$Error = T - O$$

The cost function is

$$J = 0.5(T - O)^2$$

The idea is that when training is finished, *J=0* and $\frac{\partial J}{\partial w_i} = 0$

Use this information to train.

Training is sliding down the slope of cost function

The slope of the cost function is

$$\frac{\partial J}{\partial w_i} = x_i(T - O)$$

Sliding down the slope implies that we take the negative value of the above and move in that direction

Thus

$$\Delta w_i = -x_i(T - O)$$
$$w_i^{new} = w_i^{old} + \Delta w_i$$

However, this is modified – since the problem is not a perfect quadratic.

Select $0 < \eta \leq 1$ (Learning parameter (rate))

$$\Delta w_i = -\eta x_i(T - O)$$
$$w_i^{new} = w_i^{old} + \Delta w_i$$

Okay here is the perceptron training algorithm

Computing output of perceptron

```
net_sum=0
For all i
        net_sum=net_sum+ input*weight[1]
End for
Output = activation(net_sum)
```

Perceptron Training Algorithm

```
Initialize weights randomly
While error_squared function is not equal to zero
        For each sample
        Simulate perceptron
                For all inputs i
                        Delta=T-O
                        Weights[i] += learning_rate*Delta*input[i]
                        Error=Delta
                End for
        End for
End while
```

Yes, those caveats

1. Select $0 \prec \eta \leq 1$
2. Ensure that data is representative of the system. Else you will have problems with generalisation
3. For every perceptron and weights you have there will always be another set of suitable weights – in other words your solution is “not unique”
4. There will be problems for which there is not solution.

Step 1: ***Initialisation***

Set initial weights $w_1, w_2, \ldots, w_n$ and threshold θ to random numbers in the range $[-0.5, 0.5]$.

Step 2: ***Activation***

Activate the perceptron by applying inputs $x_1(p), x_2(p), \ldots, x_n(p)$ and desired output $Y_d(p)$. Calculate the actual output at iteration $p = 1$

$$Y(p) = step\left[\sum_{i=1}^{n} x_i(p) w_i(p) - \theta\right],$$

where n is the number of the perceptron inputs, and *step* is a step activation function.

Step 3: ***Weight training***

Update the weights of the perceptron

$$w_i(p+1) = w_i(p) + \Delta w_i(p),$$

where $\Delta w_i(p)$ is the weight correction at iteration p.

The weight correction is computed by the **delta rule**:

$$\Delta w_i(p) = \alpha \times x_i(p) \times e(p)$$

Step 4: ***Iteration***

Increase iteration p by one, go back to Step 2 and repeat the process until convergence.

Epoch	Inputs x_1	Inputs x_2	Desired output Y_d	Initial weights w_1	Initial weights w_2	Actual output Y	Error e	Final weights w_1	Final weights w_2
1	0	0	0	0.3	−0.1	0	0	0.3	−0.1
	0	1	0	0.3	−0.1	0	0	0.3	−0.1
	1	0	0	0.3	−0.1	1	−1	0.2	−0.1
	1	1	1	0.2	−0.1	0	1	0.3	0.0
2	0	0	0	0.3	0.0	0	0	0.3	0.0
	0	1	0	0.3	0.0	0	0	0.3	0.0
	1	0	0	0.3	0.0	1	−1	0.2	0.0
	1	1	1	0.2	0.0	1	0	0.2	0.0
3									
4									
5									

Threshold: $\theta = 0.2$; learning rate: $\alpha = 0.1$.

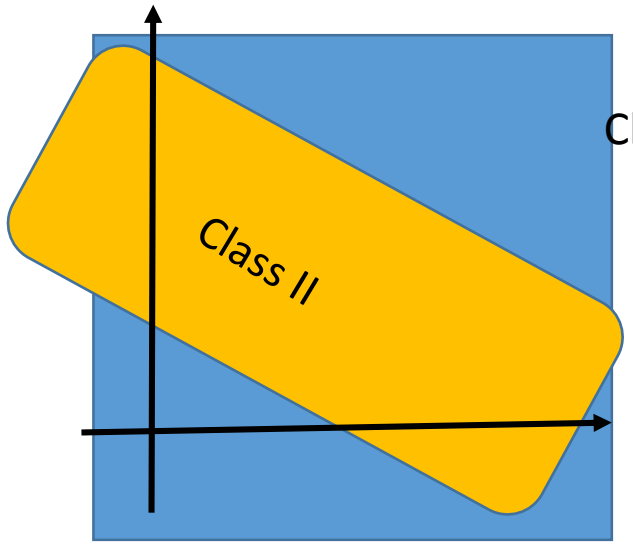
Your home work fill in the table

Source: AI by Negnevitsky, Haykin, papers and other sources on the net.

For part III of ACW – you need to use the algorithm discussed. Rather than variables x1 and x2 you need to take past samples of the location of the robot. The neuron is then giving you a model of the form:

$$y(k+1) = Sigmoid\left(\sum_{i=1}^{m} b_i y(k-i)\right)$$

You need to select first m=1, m=2, m=3 and see what you get.

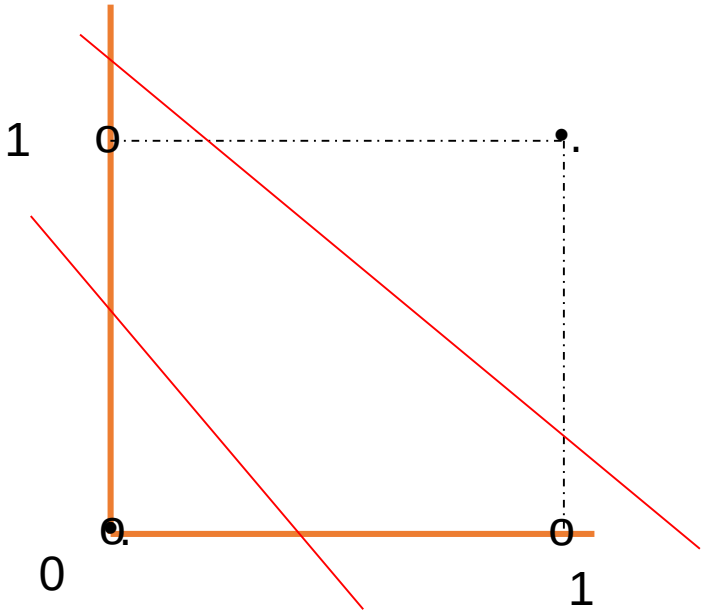


Decision surface (boundary) Complex

the XOR decision

X_1	X_2	Decision
1	0	1
1	1	0
0	1	1
0	0	0

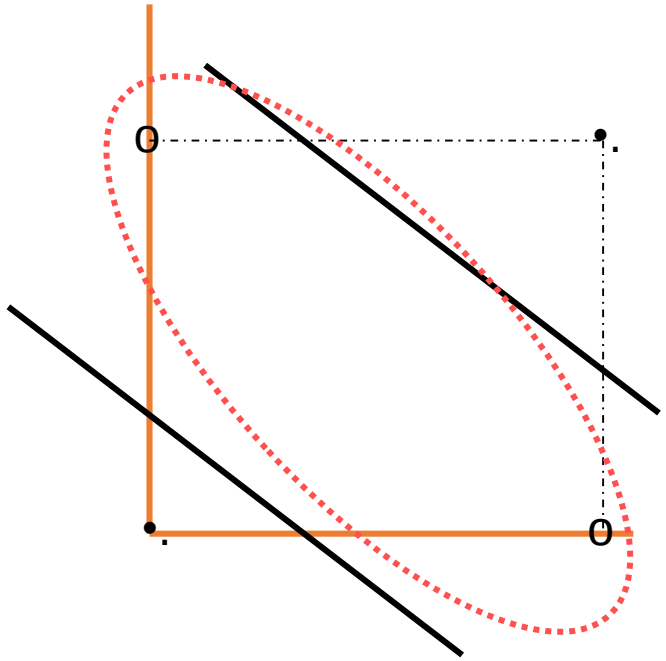
- =0 and O=1 in figure below



This requires ***2*** straight lines to separate the decision regions

Inseparable Problem ➔ Multiple decision boundaries

$$w_1 x_1 + w_2 x_2 + w_3 = 0$$
$$w_4 x_1 + w_5 x_2 + w_6 = 0$$



Two parallel lines, an ellipse or any other curve needed to solve the problem

One perceptron cannot provide this: since the decision boundary is given by

$$w_1 x_1 + w_2 x_2 + w_3 x_3 + \cdots + w_n x_n + \theta = 0$$

Lets analyse what is required : the XOR example

Need two straight lines of the form

Line 1 $\longrightarrow$ $w_{11}x_1 + w_{12}x_2 + \theta_1 = 0$

Line 2 $\longrightarrow$ $w_{21}x_1 + w_{22}x_2 + \theta_2 = 0$

The region between the lines is given by

Line3 (region) $\longrightarrow W_{31}(output1) + W_{32}(output2) + \theta_3 = 0$

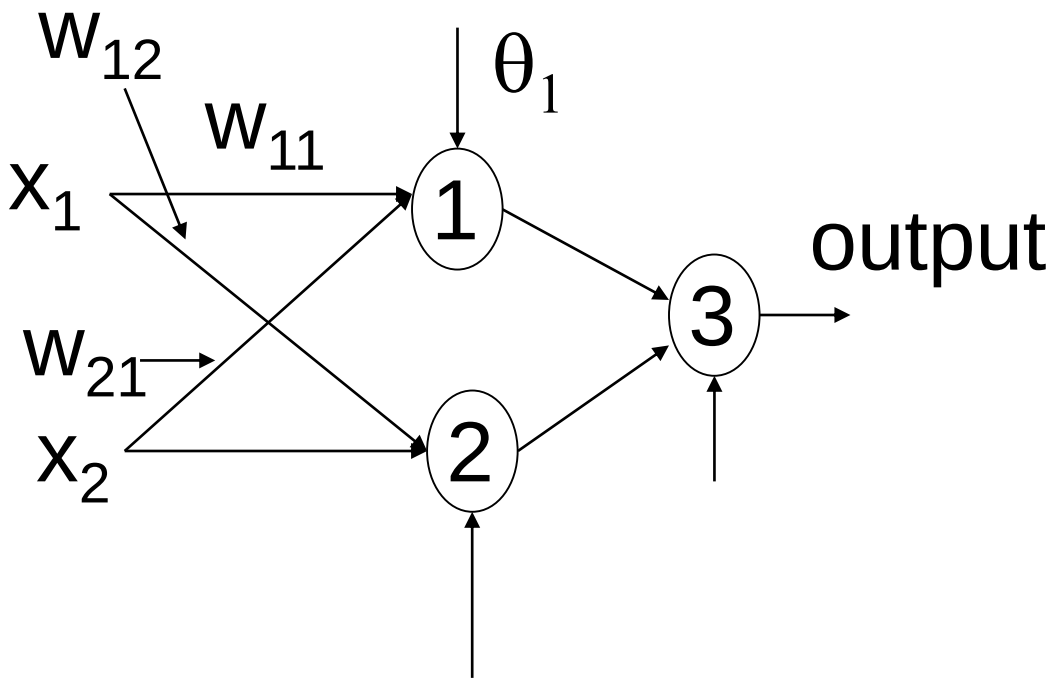
It is then clear that

(a)line3 is given by one node, line1 by another, and line2 by another node.

(b) The inputs to nodes1&2 are the input patterns, and for node3 the inputs are the outputs of nodes1&2

Okay so its 3 nodes then – so how are they connected

Lets work our way back from the output



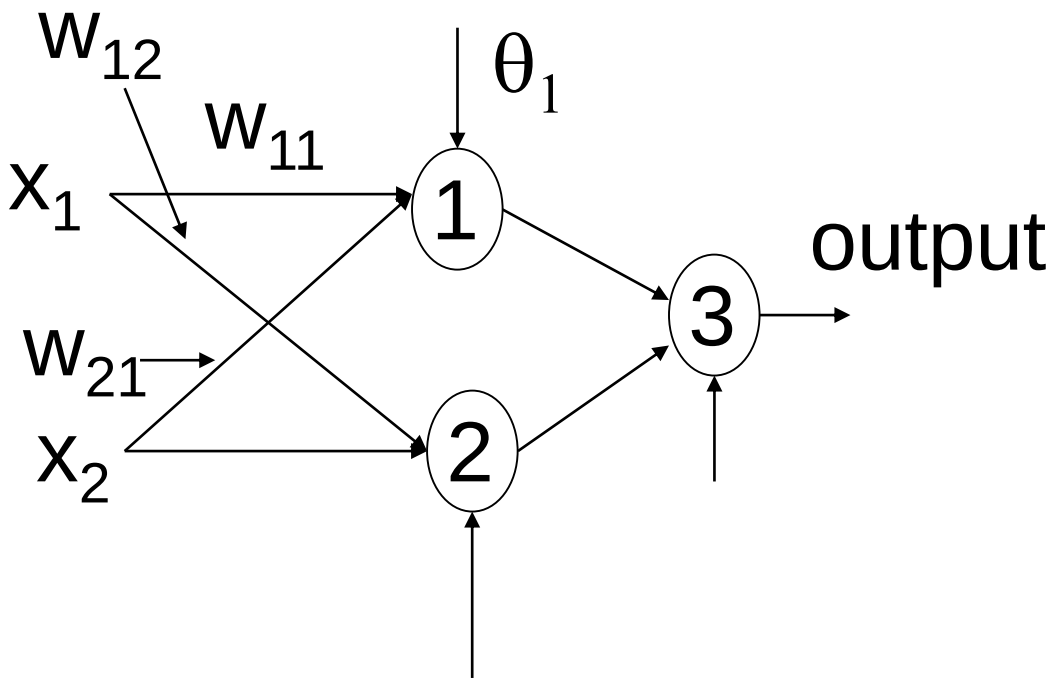
So you need to connect in series and in parallel

Note: the direction of information flow

(a)From column (layer) to the next

(b)Nodes within a column (layer) not connected

Okay here is one solution for the XOR problem



$W_{11}=w_{12}=w_{21}=w_{22}=1.0$; $\theta_1=1.5$; $\theta_2=0.5$; $\theta_3=0.5$

Here are the decision boundaries

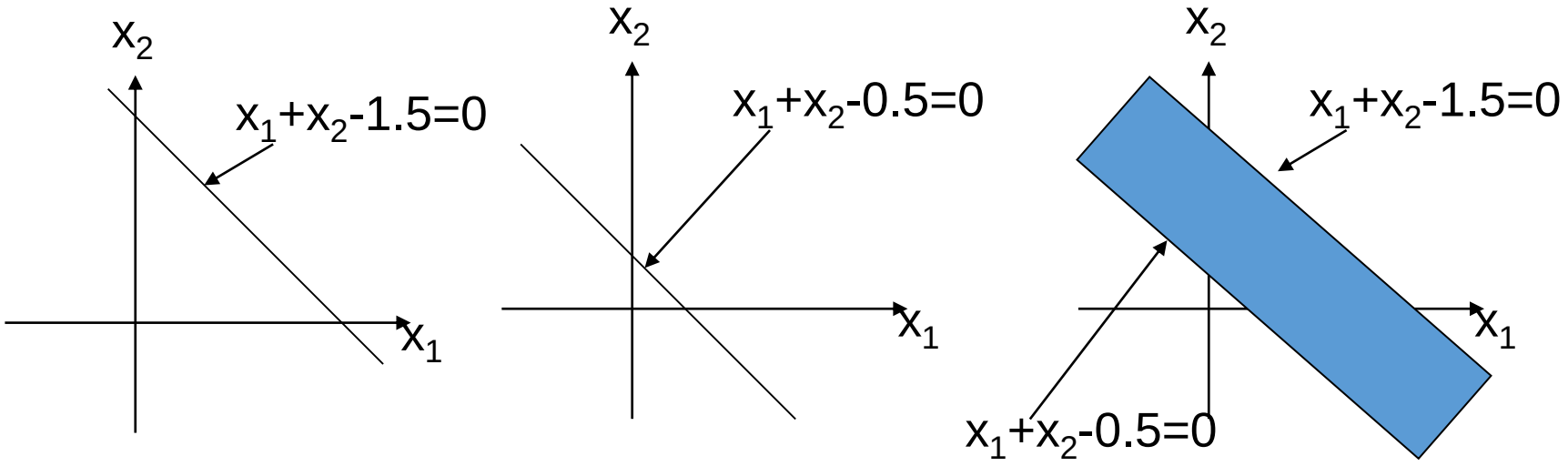


Fig a

Fig b

Fig c

Fig a : decision boundary given by node 1

Fig b: decision boundary given by node 2

Fig c: decision boundaries given by node 3

Hey, that's a multilayered perceptron

Its got three layers

(a) An input layer – nothing much happens here other than distribution of inputs
(b) A hidden layer – weighted information is transformed here to provide inputs for the next layer
(c) An output layer which transforms the outputs of the hidden layer into something meaningful to us

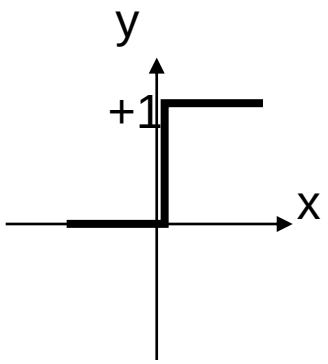
Are the nodes different in each layer

- The nodes in input layer are all linear

- The nodes in other layers are not. They are non-linear transformers – i.e nonlinear activation functions.

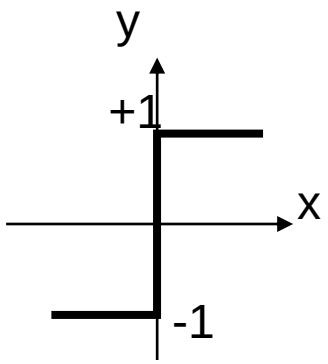
- Perceptrons – hard limiters
- In neural networks these are extended to include logistic sigmoids tanh etc.

Activation functions



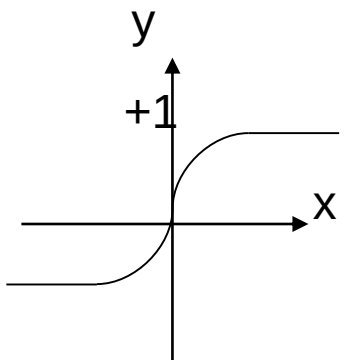
Step function

$$y = \begin{cases} 1 & if \quad x \geq 0 \\ 0 & if \quad x \prec 0 \end{cases}$$



sgn function

$$y = \begin{cases} +1 & if \quad x \geq 0 \\ -1 & if \quad x \prec 0 \end{cases}$$



sigmoid function

$$y = \frac{1}{1 + e^{-x}}$$

So which one is the preferred activation function

- In general it does not matter which activation function you use.
- However you should recall that learning or weight update uses a gradient (lecture 1) based algorithm.

- Gradient is the first derivative of the error^2 function. This implies that the activation function is differentiable.

- The step, hard limiter etc are not differentiable everywhere – there is a discontinuity (at x=0)
- Sigmoids are thus preferred since they are differentiable everywhere and in fact continuously differentiable

Derivatives of sigmoids

Apart from the fact that these s-shaped functions bear a resemblance to neuronal activation functions – they have an interesting mathematical property. You really don't need to differentiate them to get derivatives

$$S(x) = \frac{1}{1+e^{-x}}$$

$$S'(x) = -\frac{1}{(1+e^{-x})^2} \cdot (1+e^{-x})'$$

$$= -\frac{1}{(1+e^{-x})^2} \cdot (-e^{-x})$$

$$= \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}}$$

$$= S(x)(1-S(x))$$

This is useful while implementing the learning algorithms

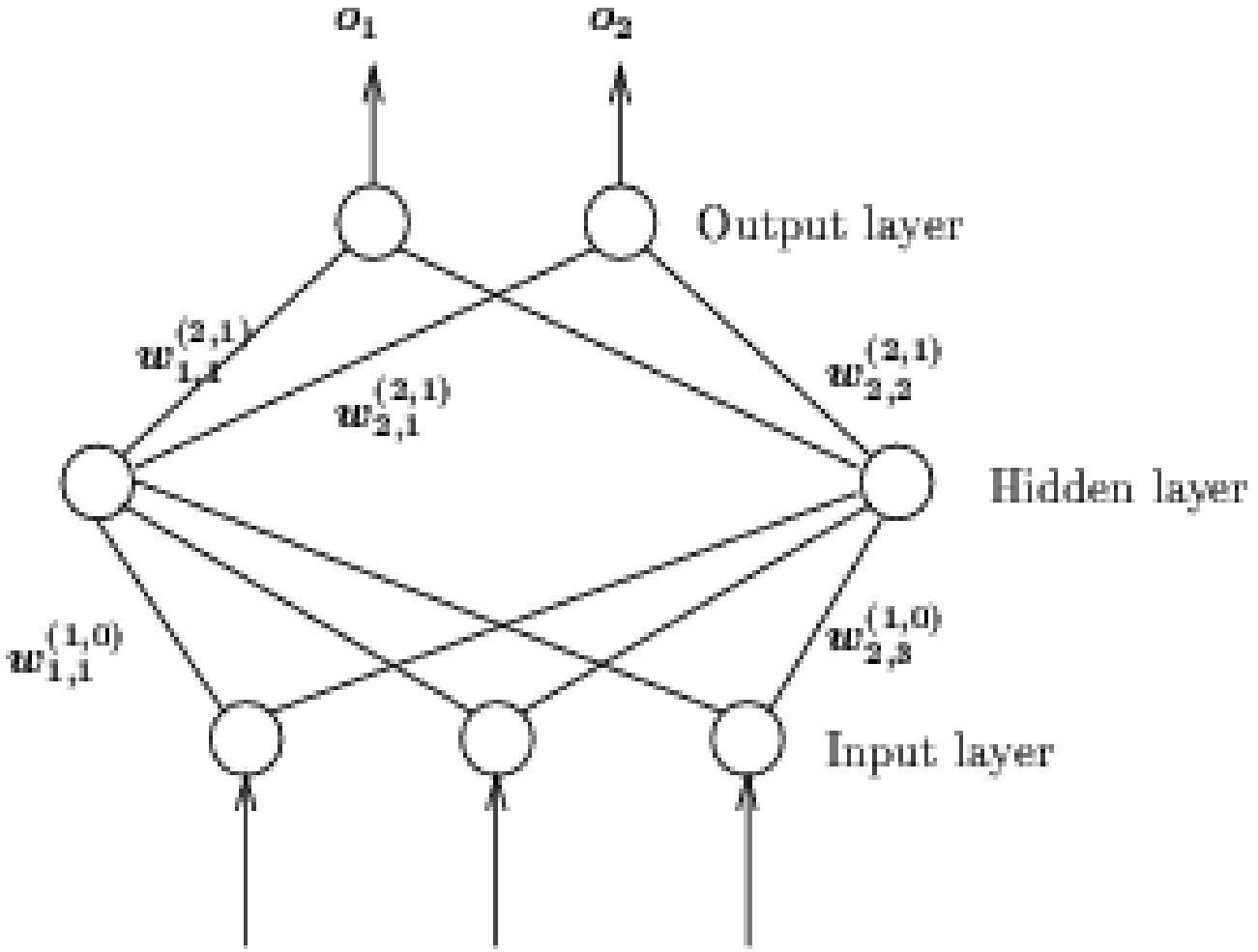
lets do the learning with back propagation

- **Architecture**:
  - **Feedforward** network of at least one layer of **non-linear** hidden nodes, e.g., # of layers ***L ≥ 2*** (not counting the input layer)
  - Node function is differentiable

    most common: **sigmoid function** $S_{net} = \frac{1}{1+e^{-net}}$
- **Learning**: supervised, error driven, generalized delta rule
- Call this type of nets BP nets
- The weight update rule (gradient descent approach)

What more! Preliminaries: information is a two way street

Forward – calculate outputs



Backward – propagate error

The algorithm for implementation

Algorithm Backpropagation;

Start with randomly chosen weights;

while MSE is unsatisfactory

and computational bounds are not exceeded, do

for each input pattern x_p, $1 \leq p \leq P$,

Compute hidden node inputs ($net_{p,j}^{(1)}$);

Compute hidden node outputs ($x_{p,j}^{(1)}$);

Compute inputs to the output nodes ($net_{p,k}^{(2)}$);

Compute the network outputs ($o_{p,k}$);

Modify outer layer weights:

$$\Delta w_{k,j}^{(2,1)} = \eta(d_{p,k} - o_{p,k})\mathcal{S}'(net_{p,k}^{(2)})x_{p,j}^{(1)}$$

Modify weights between input & hidden nodes:

$$\Delta w_{j,i}^{(1,0)} = \eta \sum_k \left((d_{p,k} - o_{p,k})\mathcal{S}'(net_{p,k}^{(2)})w_{k,j}^{(2,1)}\right) \mathcal{S}'(net_{p,j}^{(1)})x_{p,i}$$

end-for

end-while.

Note: if S is a logistic function, then $S'(x) = S(x)(1 - S(x))$