

Non-symbolic machine learning

Neural networks

- Russell and Norvig, Chapter 18 (18.7)

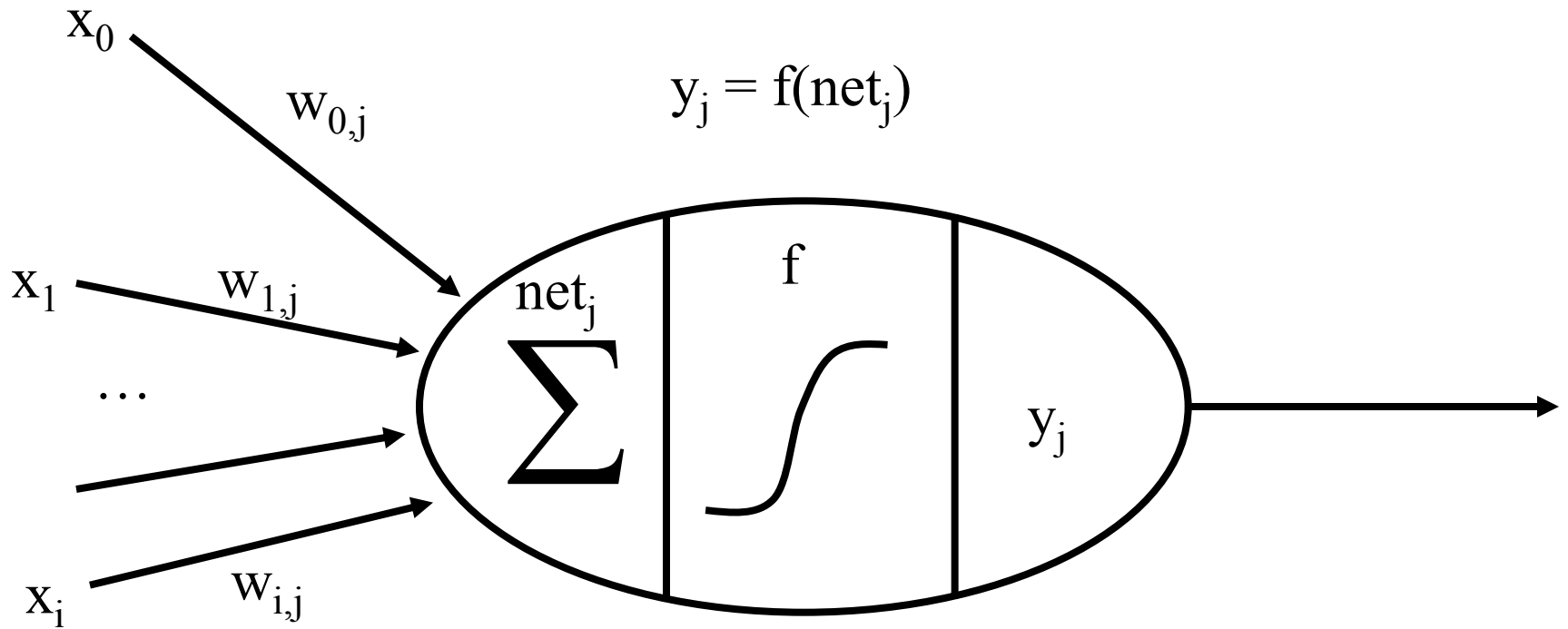
Overview: aims

- know what a neuron / unit is
- understand single-layer neural networks
- understand multi-layer neural networks
- know what a learning error is
- understand how backpropagation learning optimises weights by gradient descent
- know what a learning rate is (and how it affects learning)
- understand why/how weights and internal activations (hidden unit outputs) in the network change during learning
- have ideas of how to present data to a network
- know of the design-train-test methodology

Overview: topics

- Units
- Network
- Decision boundaries
- Multi-layer networks
- Learning in neural networks
- Gradient Descent
- Backpropagation
- Online and Batch learning
- Building a neural network

The unit



Input
Links

Input
Function

Activation
Function

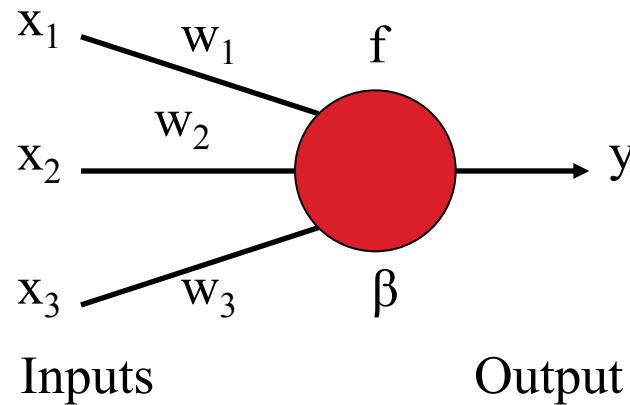
Output

Output
Links

The unit: weighting input

$$net = \sum_{i=1}^n x_i w_i + \beta$$

n inputs x_i , n weights w_i ,
bias β , output y .

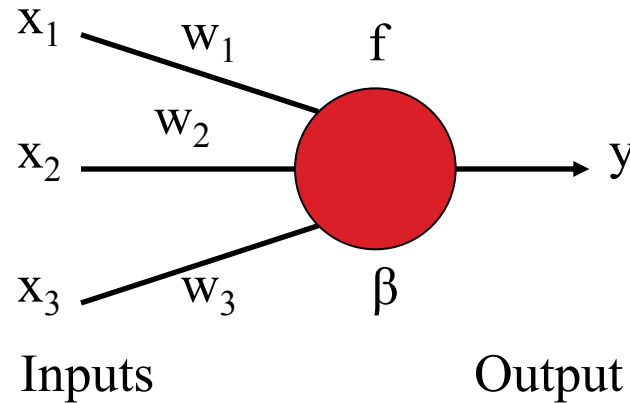


```
public double sum(double[] x, double[] w, double bias){  
    double sum=bias;  
    for (int i=0; i<x.length; i++)  
        sum+=x[i]*w[i];  
    return sum;  
}
```

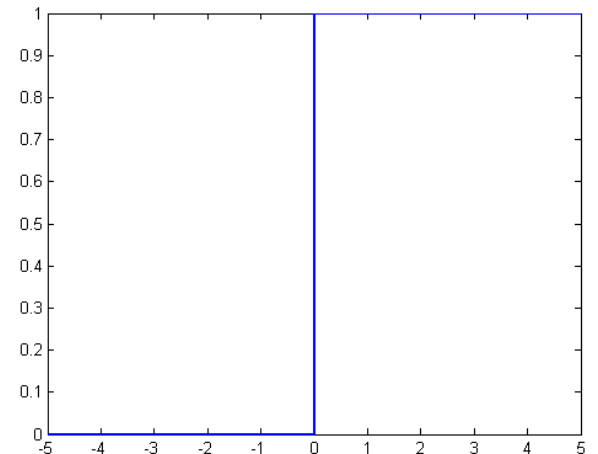
The unit: forming output: threshold

$$y = f(\text{net})$$

$$f(\text{net}) = \begin{cases} 0 & \text{if } \text{net} < 0 \\ 1 & \text{if } \text{net} \geq 0 \end{cases}$$



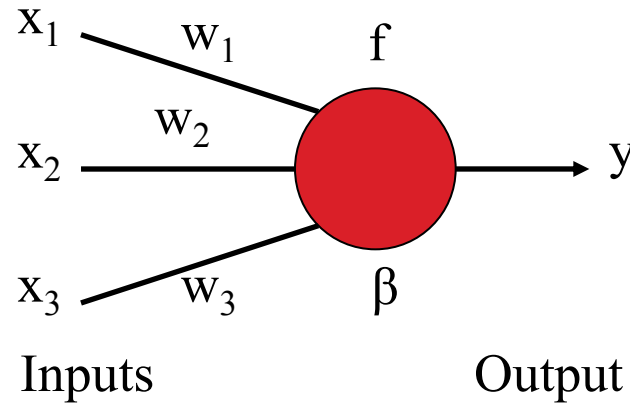
```
public double outputFunction(double net) {  
    // threshold function  
    return (net >= 0.0 ? 1.0 : 0.0);  
}
```



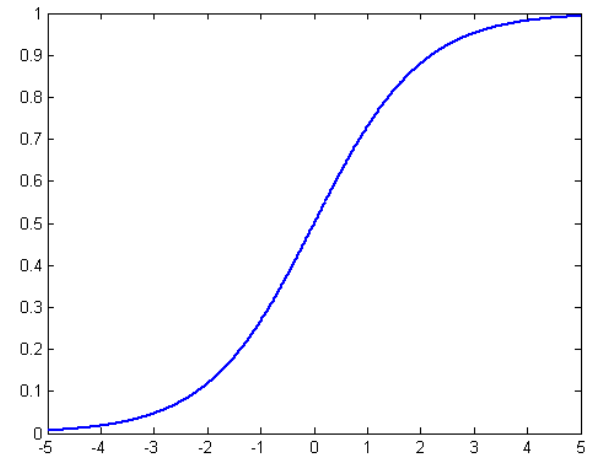
The unit: forming output: sigmoid

$$y = f(\text{net})$$

$$f(\text{net}) = \frac{1}{1 + e^{-\text{net}}}$$



```
public double outputFunction(double net) {  
    // a sigmoid function  
    return 1.0 / (1.0 + Math.exp(-net));  
}
```



What can a unit compute?

- Regression
 - E.g. continuous output
- Classification
 - E.g. logic functions (AND, OR, NOT etc.)

Logic functions

$$Y = X1 \text{ OR } X2$$

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	1

$$Y = X1 \text{ AND } X2$$

X1	X2	Y
0	0	0
0	1	0
1	0	0
1	1	1

$$Y = X1 \text{ NOR } X2$$

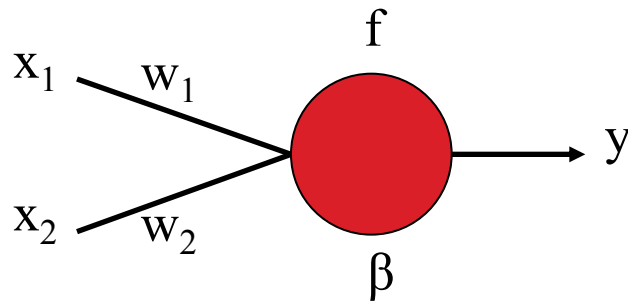
X1	X2	Y
0	0	1
0	1	0
1	0	0
1	1	0

$$Y = X1 \text{ NAND } X2$$

X1	X2	Y
0	0	1
0	1	1
1	0	1
1	1	0

Logic functions: OR (1)

	X1	X2	Y
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	1



$$x_1w_1 + x_2w_2 + \beta = net$$

$$1. w_1 * 0 + w_2 * 0 + \beta < 0$$

$$\beta < 0$$

$$2. w_1 * 0 + w_2 * 1 + \beta \geq 0$$

$$w_2 + \beta \geq 0$$

$$w_2 \geq -\beta$$

$$3. w_1 * 1 + w_2 * 0 + \beta \geq 0$$

$$w_1 + \beta \geq 0$$

$$w_1 \geq -\beta$$

$$4. w_1 * 1 + w_2 * 1 + \beta \geq 0$$

$$w_1 + w_2 + \beta \geq 0$$

$$w_1 + w_2 \geq -\beta$$

$$net = \sum_{i=1}^n x_i w_i + \beta$$

$$= x_1 w_1 + x_2 w_2 + \beta$$

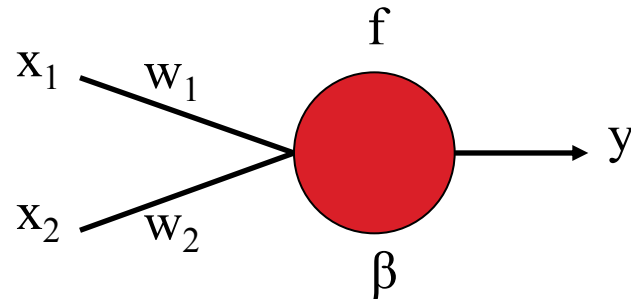
$$f(net) = \begin{cases} 0 & \text{if } net < 0 \\ 1 & \text{if } net \geq 0 \end{cases}$$

Logic functions: OR (2)

$$net = \sum_{i=1}^n x_i w_i + \beta$$

$$= x_1 w_1 + x_2 w_2 + \beta$$

$$f(net) = \begin{cases} 0 & \text{if } net < 0 \\ 1 & \text{if } net \geq 0 \end{cases}$$



$$w_1 = 1.0$$

$$w_2 = 1.0$$

$$\beta = -0.5$$

x1	x2	net	f(net)
0	0	-0.5	0
0	1	0.5	1
1	0	0.5	1
1	1	1.5	1

Decision boundary: OR

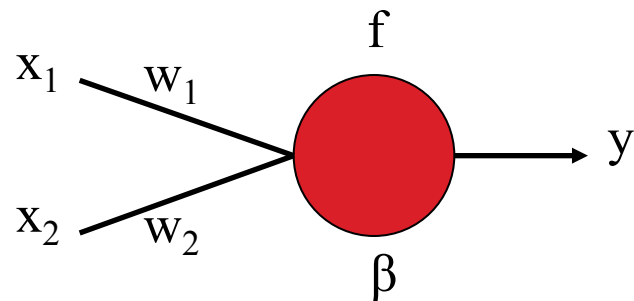
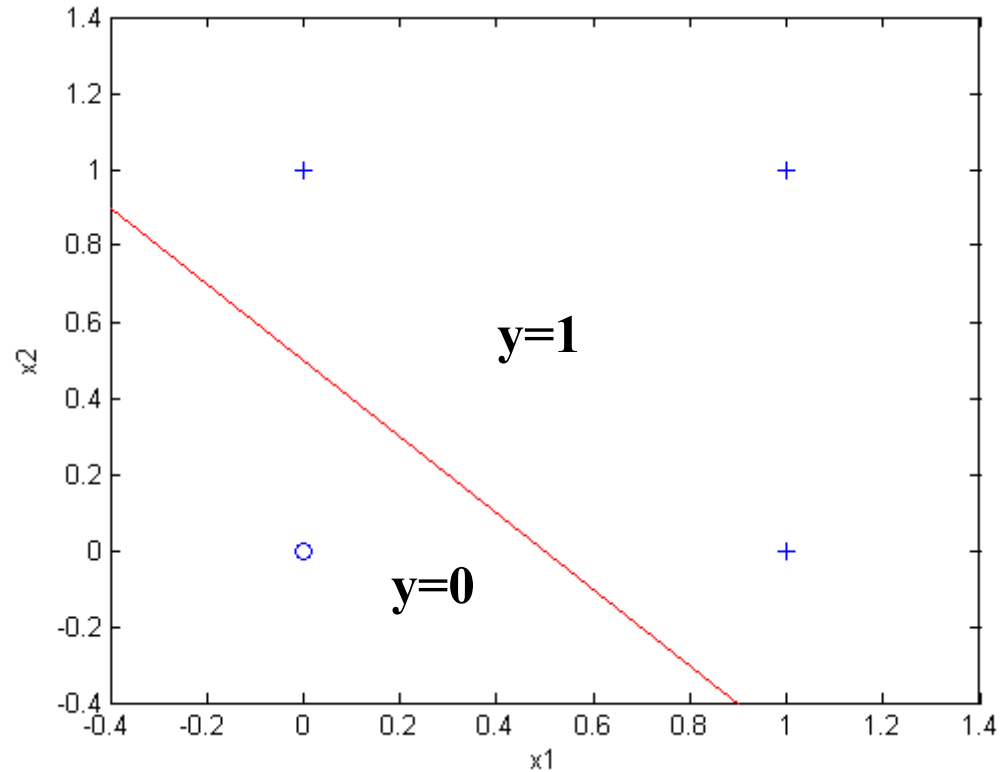
$$w_1 = 1.0$$

$$w_2 = 1.0$$

$$\beta = -0.5$$

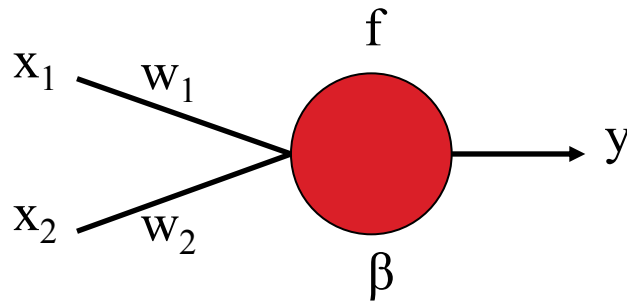
$$x_1 = -x_2 \frac{w_2}{w_1} - \frac{\beta}{w_1}$$

$$x_1 = -x_2 + 0.5$$



Logic functions: AND (1)

	X1	X2	Y
1	0	0	0
2	0	1	0
3	1	0	0
4	1	1	1



$$x_1w_1 + x_2w_2 + \beta = net$$

$$1. w_1 * 0 + w_2 * 0 + \beta < 0$$

$$\beta < 0$$

$$2. w_1 * 0 + w_2 * 1 + \beta < 0$$

$$w_2 + \beta < 0$$

$$w_2 < -\beta$$

$$3. w_1 * 1 + w_2 * 0 + \beta < 0$$

$$w_1 + \beta < 0$$

$$w_1 < -\beta$$

$$4. w_1 * 1 + w_2 * 1 + \beta \geq 0$$

$$w_1 + w_2 + \beta \geq 0$$

$$w_1 + w_2 \geq -\beta$$

$$net = \sum_{i=1}^n x_i w_i + \beta$$

$$= x_1 w_1 + x_2 w_2 + \beta$$

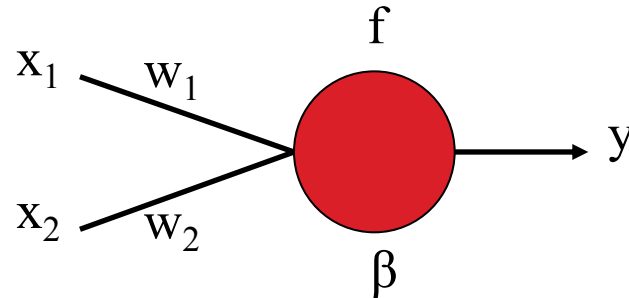
$$f(net) = \begin{cases} 0 & \text{if } net < 0 \\ 1 & \text{if } net \geq 0 \end{cases}$$

Logic functions: AND (2)

$$net = \sum_{i=1}^n x_i w_i + \beta$$

$$= x_1 w_1 + x_2 w_2 + \beta$$

$$f(net) = \begin{cases} 0 & \text{if } net < 0 \\ 1 & \text{if } net \geq 0 \end{cases}$$



$$w_1 = 1.0$$

$$w_2 = 1.0$$

$$\beta = -1.5$$

x1	x2	net	f(net)
0	0	-1.5	0
0	1	-0.5	0
1	0	-0.5	0
1	1	0.5	1

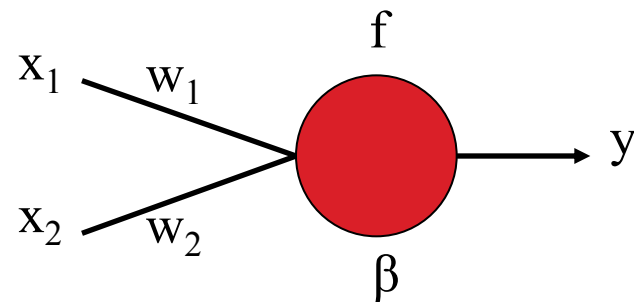
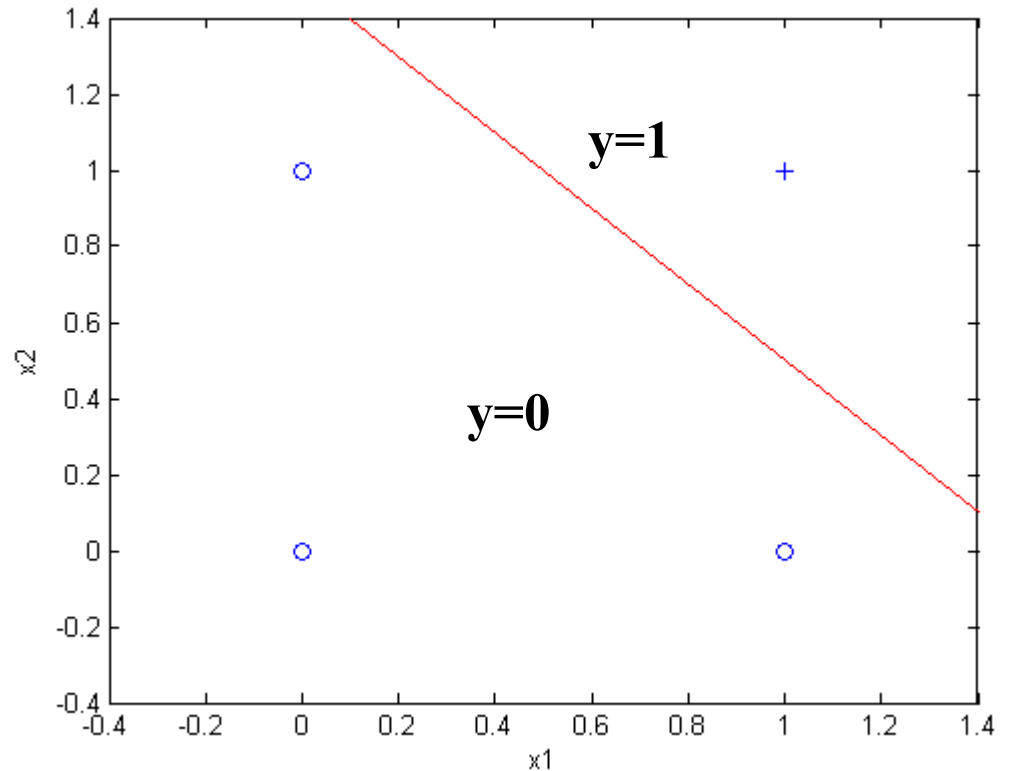
Decision boundary: AND

$$w_1 = 1.0$$

$$w_2 = 1.0$$

$$\beta = -1.5$$

$$x_1 = -x_2 \frac{w_2}{w_1} - \frac{\beta}{w_1}$$
$$x_1 = -x_2 + 1.5$$

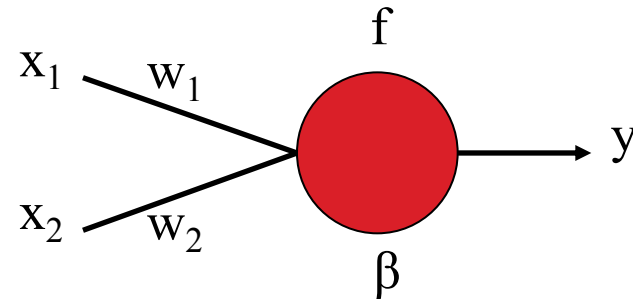


Logic functions: NOR

$$net = \sum_{i=1}^n x_i w_i + \beta$$

$$= x_1 w_1 + x_2 w_2 + \beta$$

$$f(net) = \begin{cases} 0 & \text{if } net < 0 \\ 1 & \text{if } net \geq 0 \end{cases}$$



$$w_1 = -1.0$$

$$w_2 = -1.0$$

$$\beta = 0.5$$

x_1	x_2	net	$f(\text{net})$
0	0	0.5	1
0	1	-0.5	0
1	0	-0.5	0
1	1	-1.5	0

Decision boundary: NOR

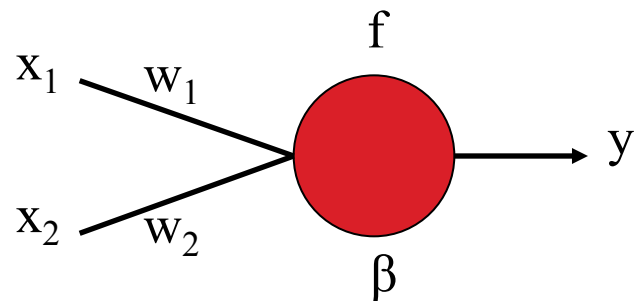
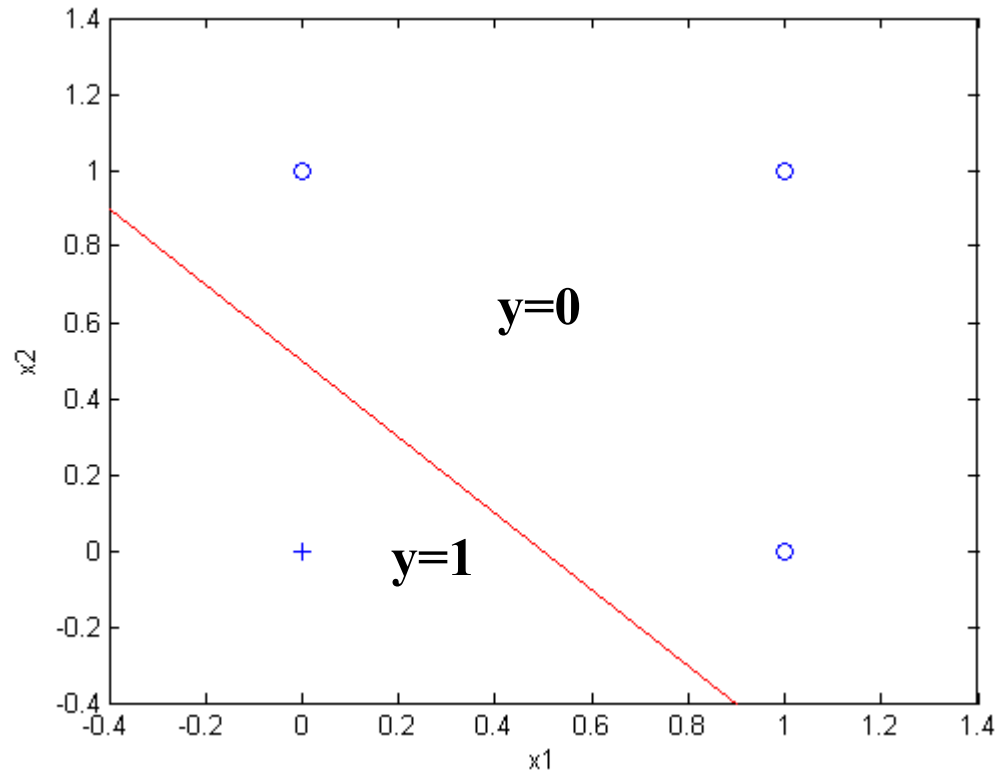
$$w_1 = -1.0$$

$$w_2 = -1.0$$

$$\beta = 0.5$$

$$x_1 = -x_2 \frac{w_2}{w_1} - \frac{\beta}{w_1}$$

$$x_1 = -x_2 + 0.5$$

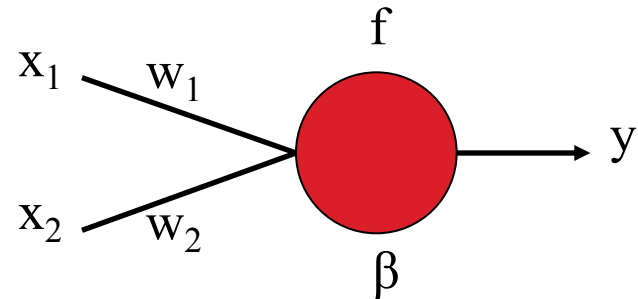


Logic functions: NAND

$$net = \sum_{i=1}^n x_i w_i + \beta$$

$$= x_1 w_1 + x_2 w_2 + \beta$$

$$f(net) = \begin{cases} 0 & \text{if } net < 0 \\ 1 & \text{if } net \geq 0 \end{cases}$$



$$w_1 = -1.0$$

$$w_2 = -1.0$$

$$\beta = 1.5$$

x1	x2	net	f(net)
0	0	1.5	1
0	1	0.5	1
1	0	0.5	1
1	1	-0.5	0

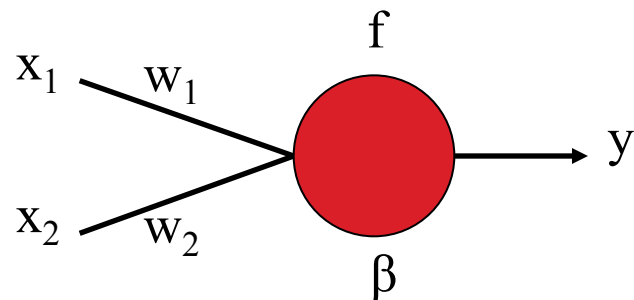
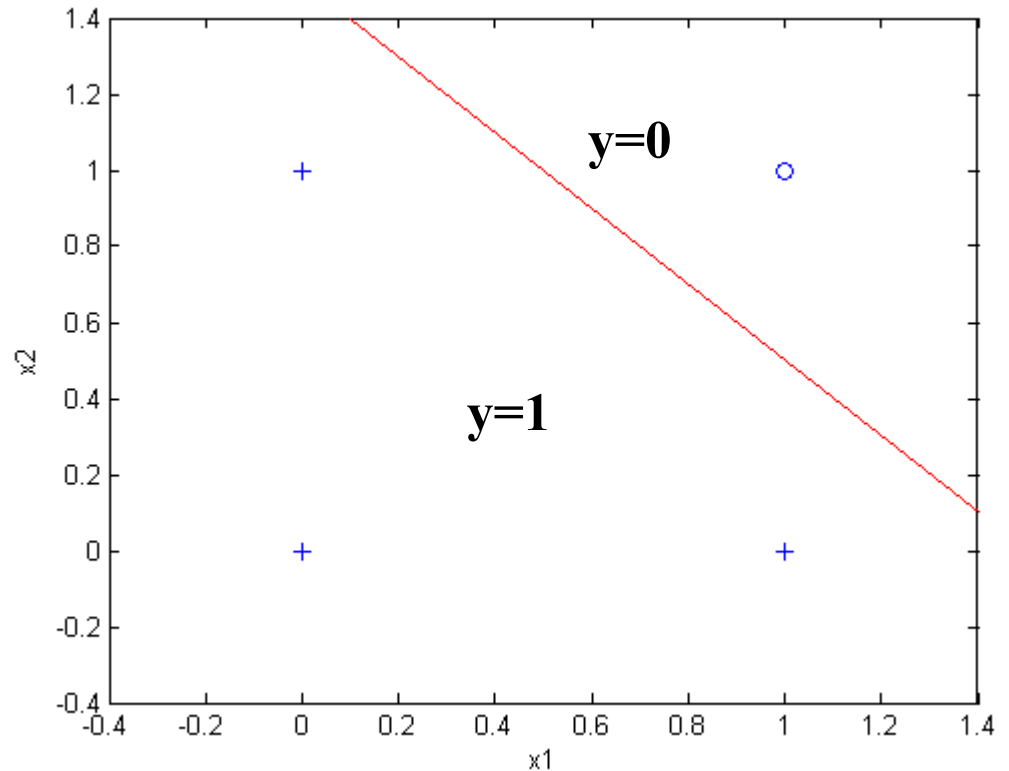
Decision boundary: NAND

$$w_1 = -1.0$$

$$w_2 = -1.0$$

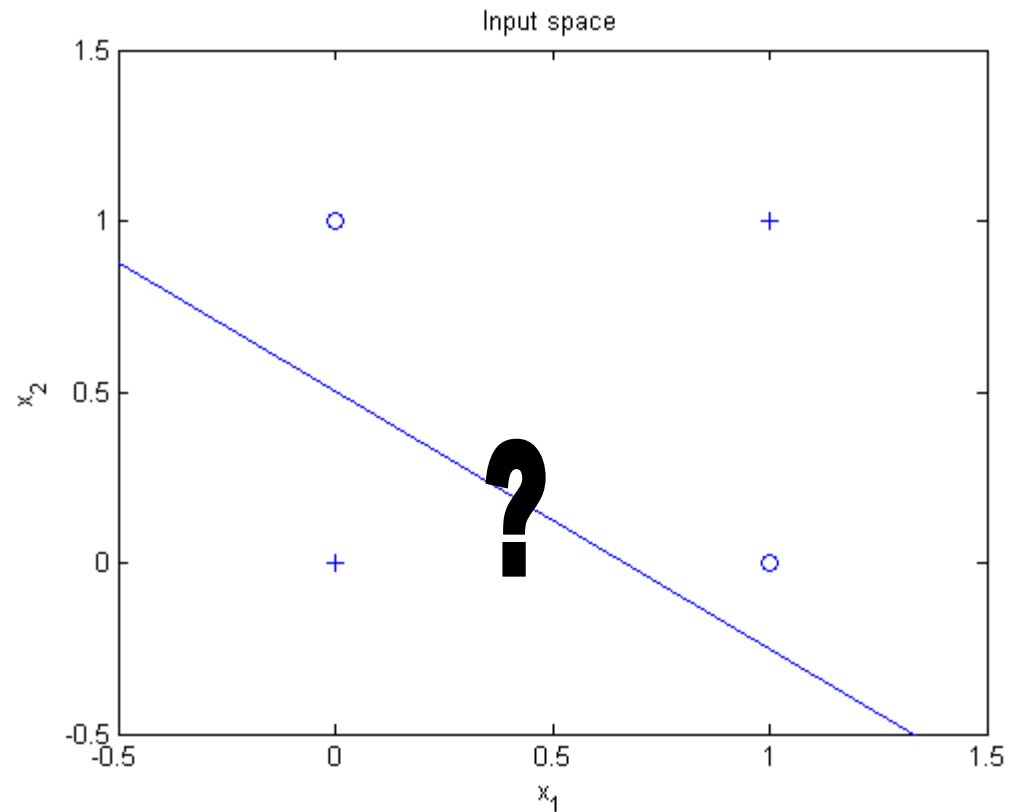
$$\beta = 1.5$$

$$x_1 = -x_2 \frac{w_2}{w_1} - \frac{\beta}{w_1}$$
$$x_1 = -x_2 + 1.5$$



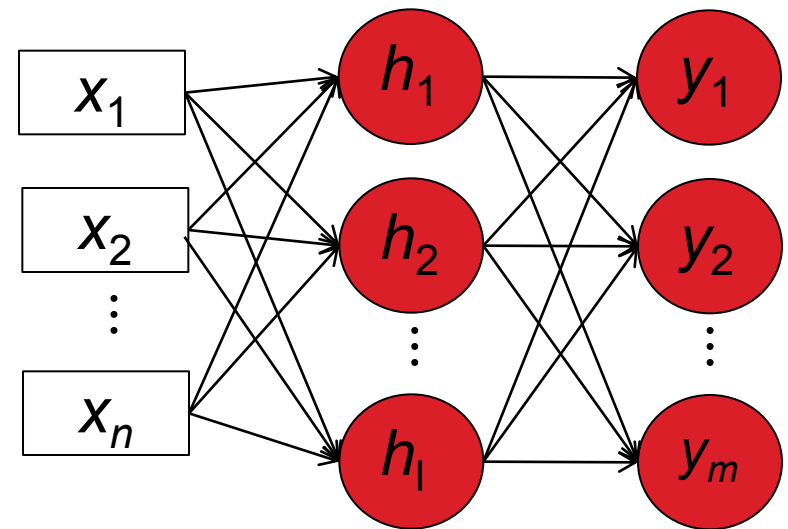
Decision boundary: XOR

- When inputs are not linearly separable, what can be done?
- Example: XOR problem:
 $y = \text{XOR}(x_1, x_2)$
- Single layer networks can only represent linearly separable functions



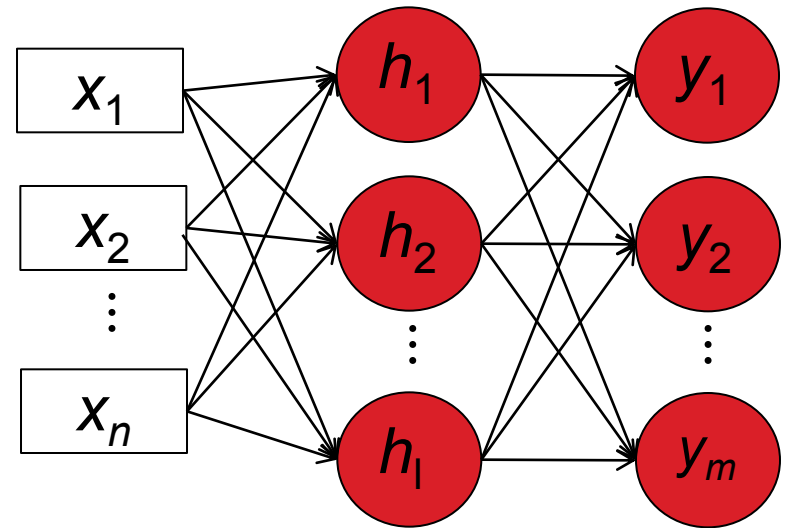
Multi-layer network

- Activation flows from input to output through a set of intermediate hidden (or latent) nodes
- The activations of the hidden nodes are decided internally by the network
- They depend upon the weights between input and hidden nodes



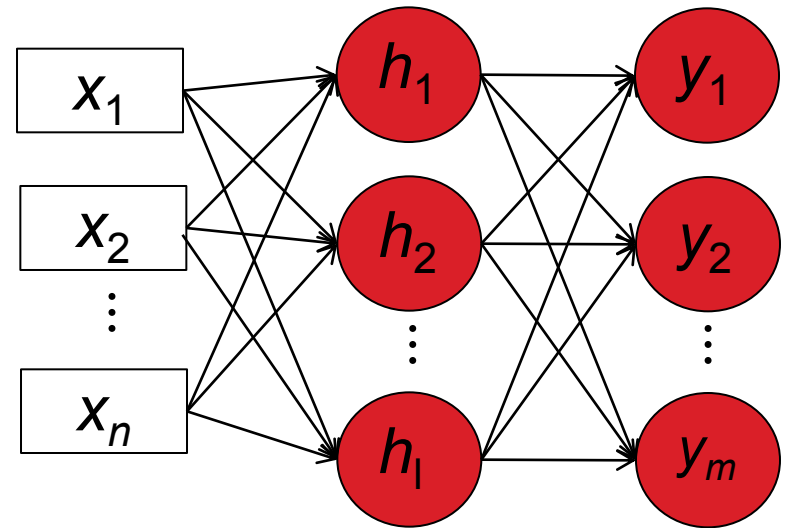
How does that work ?

- What is really happening is that the inputs are being transformed via input-hidden weights into a new space
- This space may have more or less dimensions
- We are hoping to choose weights so that the problem to solve at the hidden-output layer is then linearly separable



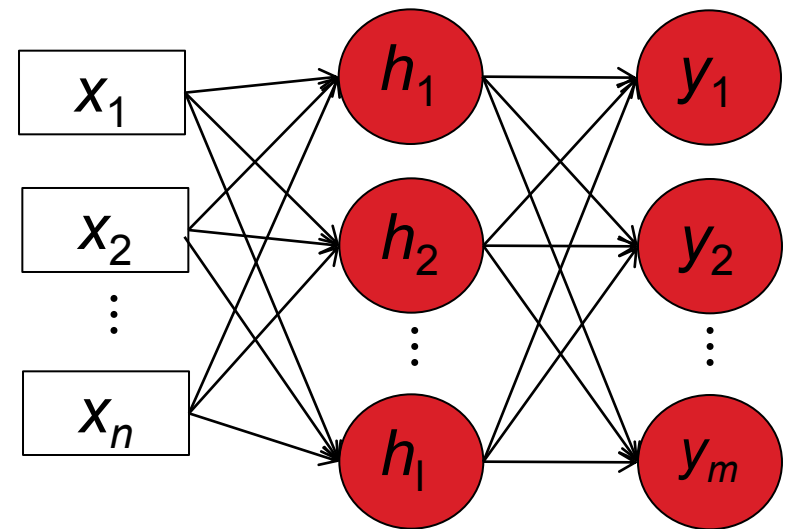
What needs to be chosen ?

- Whether to have a hidden layer or not (generally more capable with one)
- How many hidden units to use?
- Values for each of the weights (and biases) to solve our problem of interest
- Note: can use more than one hidden layer, but not usually beneficial



How to make these choices ?

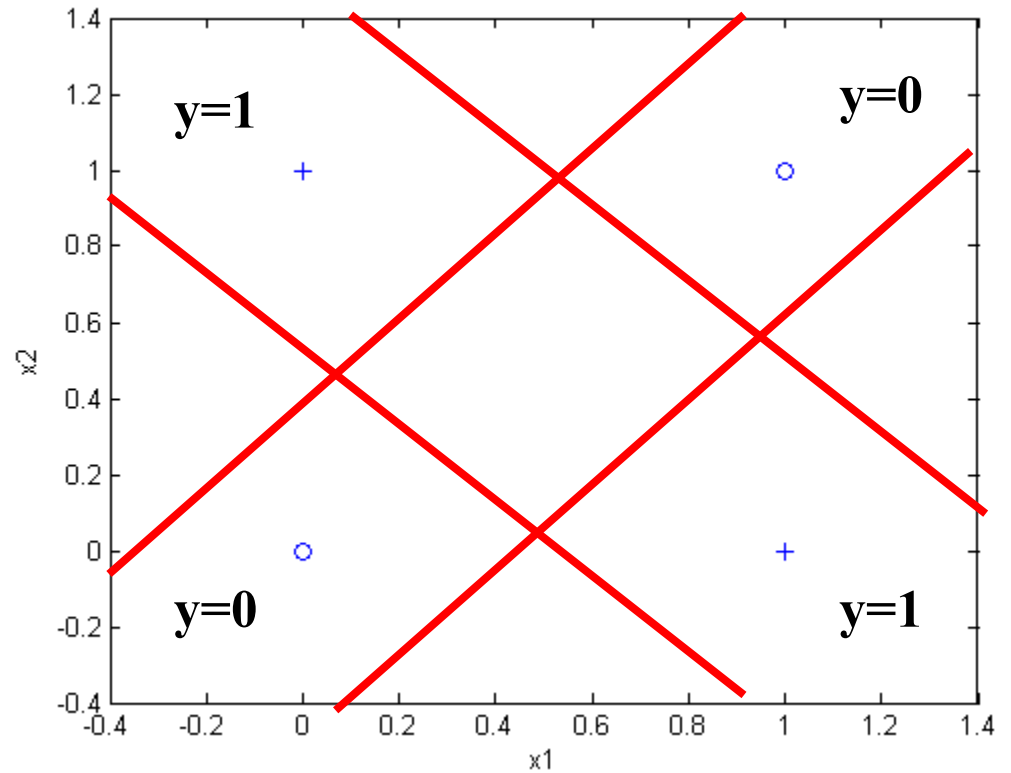
- Choose to have a hidden layer – might as well if using neural networks
- Number of hidden units: can start at 1 and add 1 at a time, comparing by somehow estimating performance
- Or can start with a big hidden layer and prune down
- Can consider pruning inputs (variable selection) and individual weights also



Solving XOR (1)

$$Y = X1 \text{ XOR } X2$$

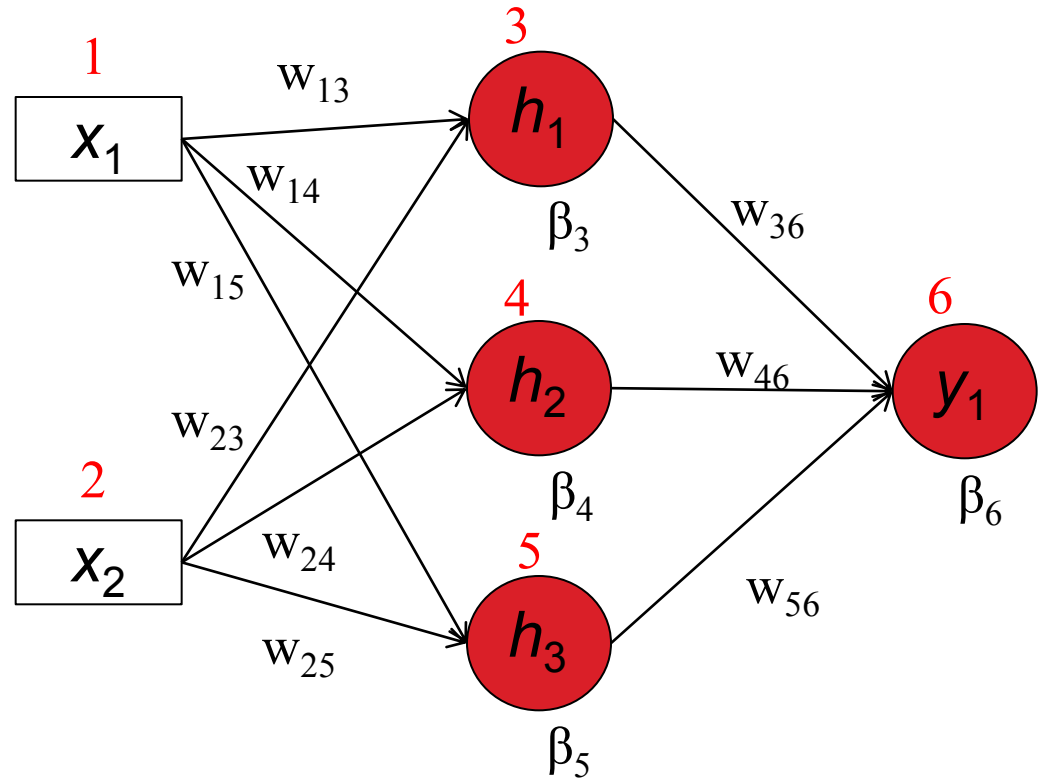
X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0



Solving XOR (2)

$$Y = X_1 \text{ XOR } X_2$$

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

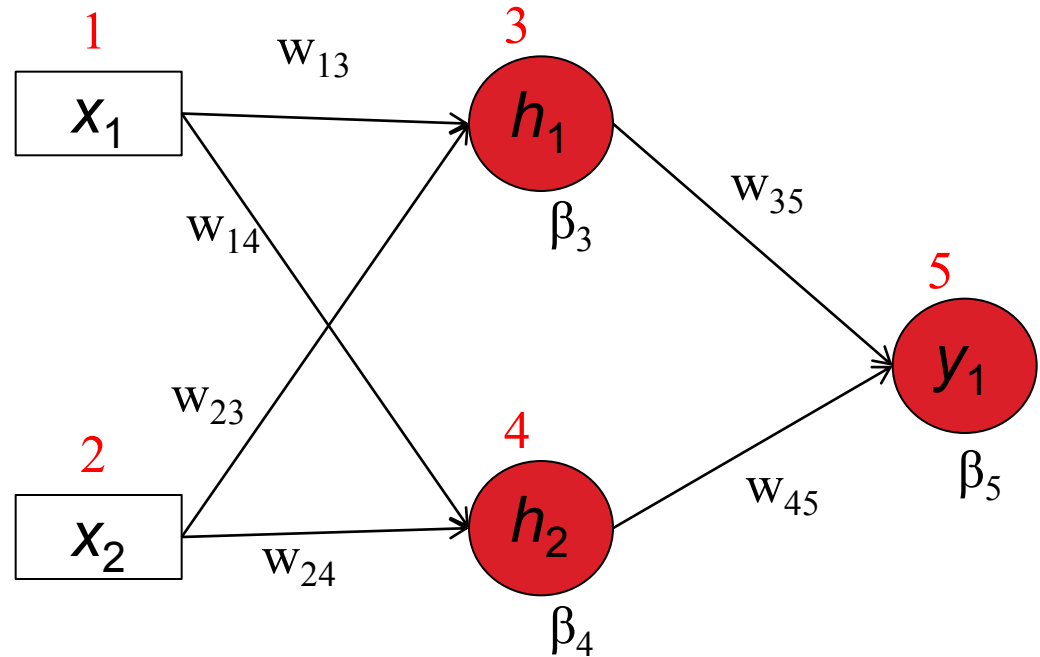


- Assume each node implements the threshold function
- Using a hidden layer, find values for the weights and biases to implement XOR
- Can be solved with integer weights

Solving XOR (3)

$$Y = X_1 \text{ XOR } X_2$$

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

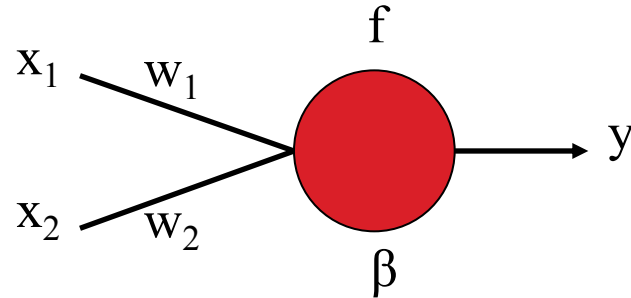


- Assume each node implements the threshold function
- Using a hidden layer, find values for the weights and biases to implement XOR
- Can be solved with integer weights

Logic functions: OR (Review)

$$net = \sum_{i=1}^n x_i w_i + \beta$$

$$f(net) = \begin{cases} 0 & \text{if } net < 0 \\ 1 & \text{if } net \geq 0 \end{cases}$$



$$w_1 = 1.0$$

$$w_2 = 1.0$$

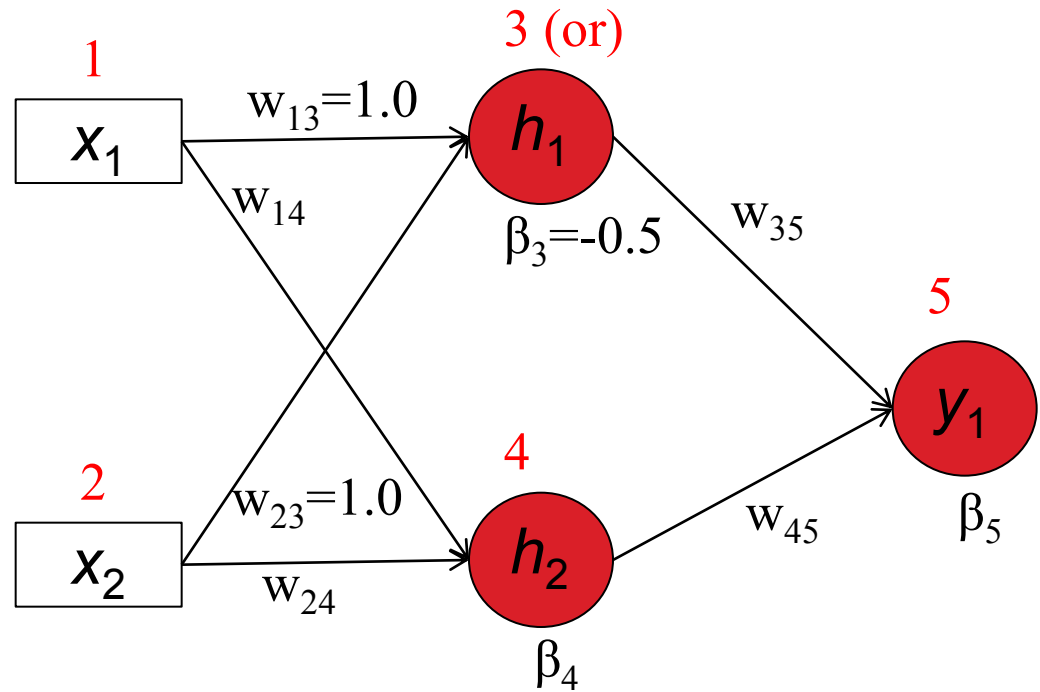
$$\beta = -0.5$$

x1	x2	net	f(net)
0	0	-0.5	0
0	1	0.5	1
1	0	0.5	1
1	1	1.5	1

Solving XOR (4)

$$Y = X1 \text{ XOR } X2$$

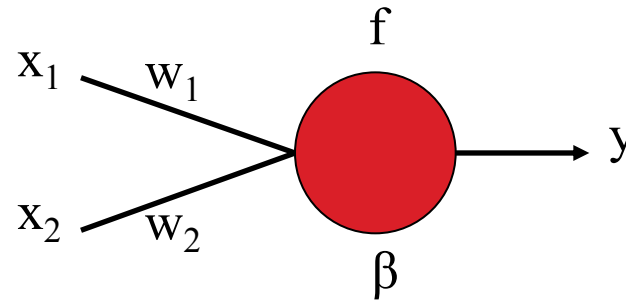
X1	X2	H1	Y
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0



Logic functions: AND (Review)

$$net = \sum_{i=1}^n x_i w_i + \beta$$

$$f(net) = \begin{cases} 0 & \text{if } net < 0 \\ 1 & \text{if } net \geq 0 \end{cases}$$



$$w_1 = 1.0$$

$$w_2 = 1.0$$

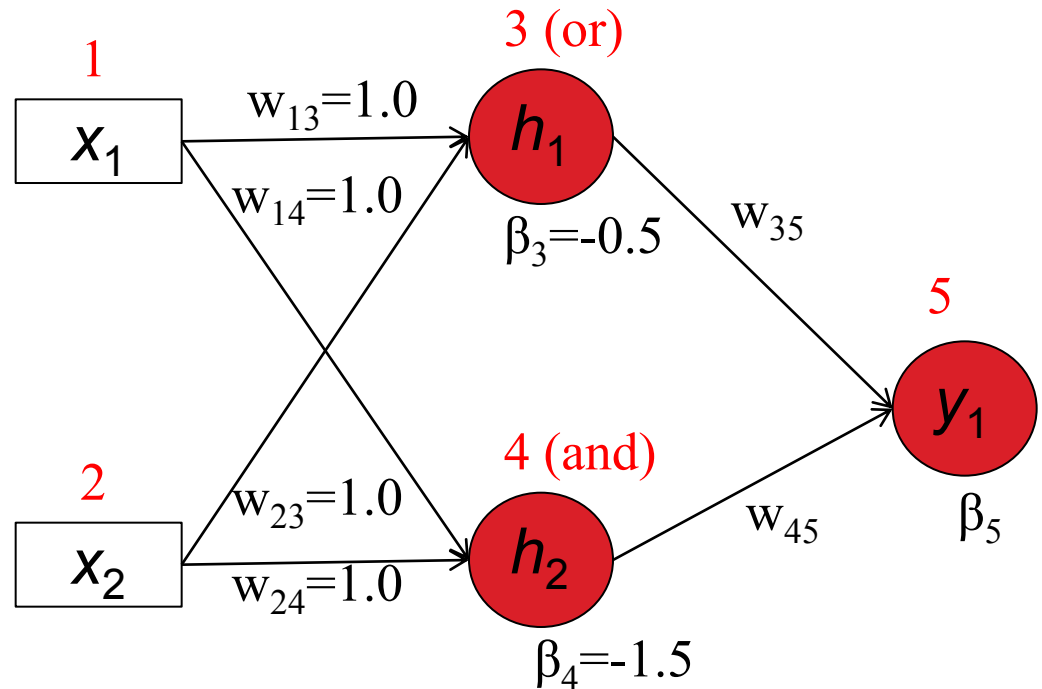
$$\beta = -1.5$$

x1	x2	net	f(net)
0	0	-1.5	0
0	1	-0.5	0
1	0	-0.5	0
1	1	0.5	1

Solving XOR (5)

$$Y = X1 \text{ XOR } X2$$

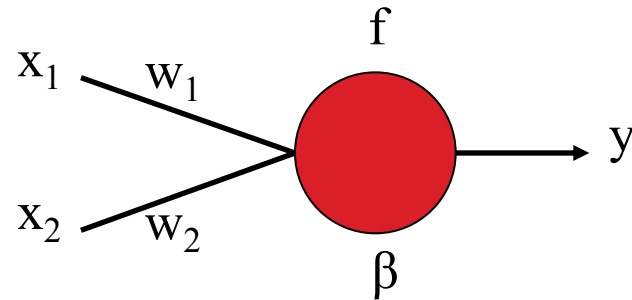
X1	X2	H1	H2	Y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0



Logic functions: x1 AND NOT x2

$$net = \sum_{i=1}^n x_i w_i + \beta$$

$$f(net) = \begin{cases} 0 & \text{if } net < 0 \\ 1 & \text{if } net \geq 0 \end{cases}$$



$$w_1 = 1.0$$

$$w_2 = -1.0$$

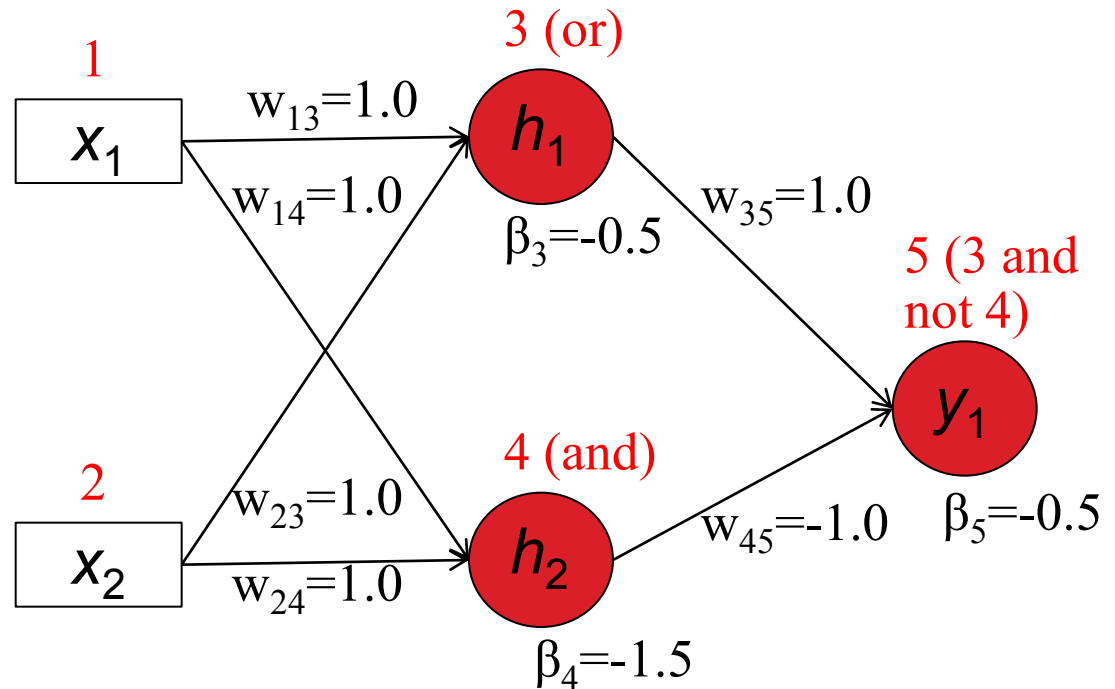
$$\beta = -0.5$$

x1	x2	net	f(net)
0	0	-0.5	0
0	1	-1.5	0
1	0	0.5	1
1	1	-0.5	0

Solving XOR (6)

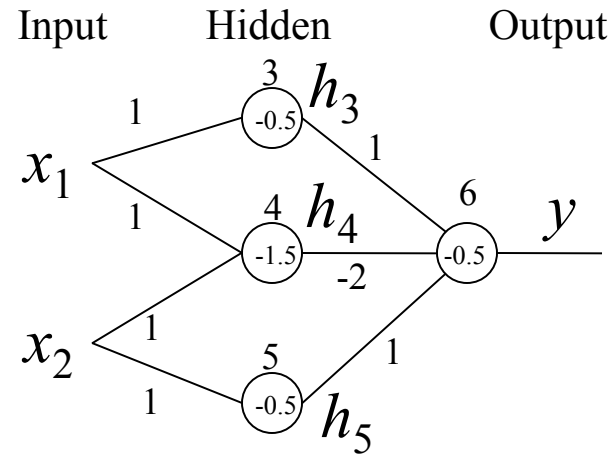
$$Y = X1 \text{ XOR } X2$$

X1	X2	H1	H2	Y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

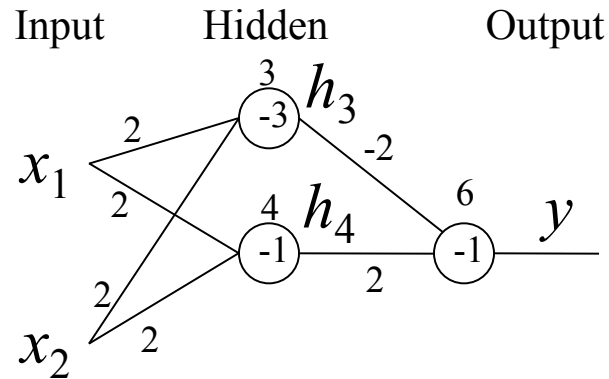


2 solutions for XOR

- 1st solution: nodes 3 and 5 just copy x_1 and x_2 through.
- Node 4 does x_1 AND x_2 .
- Result is 1 if x_1 OR x_2 , but not if x_1 AND x_2 .



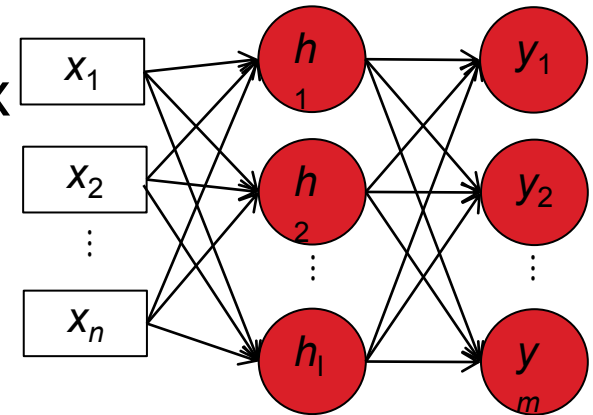
- Node 3 does AND, node 4 does OR.
- Node 6 calculates OR – AND, and the threshold only allows XOR through.



Decision boundaries with a hidden layer

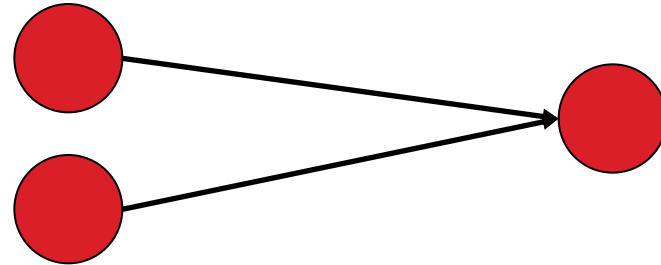
- If you were to look at the new decision boundaries in x , they would be complex curves
- Example: predicted presence and absence of tsetse flies in Zimbabwe using a neural network with 24 hidden neurons
- From Gettinby et al

<http://www.fao.org/wairdocs/ilri/x5441e/x5441e0a.htm>

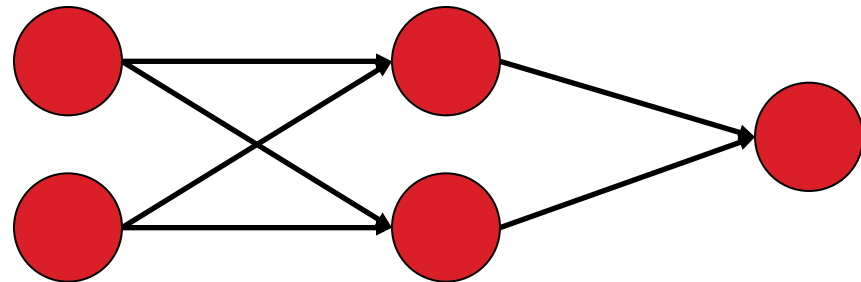


Neural Network Structures

- Feedforward
 - Single layer
 - E.g. Perceptron

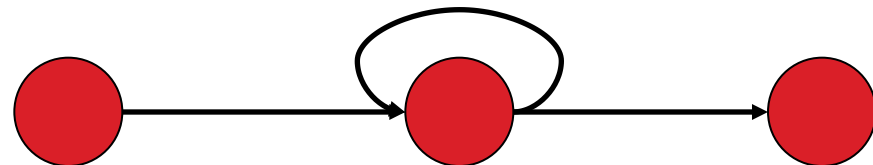


- Multi layer



- Recurrent

- Any neural network with at least one feedback connection



Learning in neural networks

- Single-layer networks
- Multi-layer networks
- Outputs:
 - Binary output: function = threshold
 - Continuous output: threshold = sigmoid

Bias weights

- Rather than use separate notation for bias weights, we use w_{0b} for the bias weight on a hidden layer unit and w_{0c} for the bias weight on an output unit
- To make this work, we assume that there is an extra input x_{i0} and an extra hidden unit h_{i0} , both of which are permanently fixed at +1

Learning in Neural Networks

- For a single layer feedforward neural network, learning can be seen as shifting the decision boundary until the training set examples are classified correctly
- Decision boundary is shifted by changing the weights (including the bias)

$$w_{jk,new} = w_{jk} + \Delta w_{jk}$$

Notation: counts and indices

- Textbook is not consistent: uses n for both number of inputs and number of training patterns.
- Earlier slides not all consistent either.
- Hereon in slides, will use:
 - Observations/patterns: $i=1,\dots,n$.
 - Training patterns: $i=1,\dots,n_{tr}$
 - Test patterns: $i=1,\dots,n_{te}$
 - If using all observations for training, $n_{tr} = n$. If test set is a subset of the observations, $n_{tr} + n_{te} = n$.
 - Inputs/attributes: $j=1,\dots,p$.
 - Outputs/responses: $k=1,\dots,m$.
 - Hidden units: $q=1,\dots,l$. (that's a lowercase L).

Notation: data and network output (1)

- Observations/patterns:
 - input: $x_i = \{x_{ij}\}$, $i = \text{pattern \#}$, $j = \text{input \#}$
 - correct output: $y_i = \{y_{ik}\}$, $i = \text{pattern \#}$, $k = \text{output \#}$
- Neural network:
 - hidden unit value: $h_i = \{h_{iq}\}$, $i = \text{pattern \#}$, $q = \text{hidden unit \#}$
 - output unit value: $o_i = \{o_{ik}\}$, $i = \text{pattern \#}$, $k = \text{output \#}$
 - Also sometimes write o_{ik} as $f_{network,k}(x_i)$ to indicate that the neural network output is a function of the input.

Notation: data and network output (2)

- weight on connection from j th input to q th hidden unit: w_{jq}
- weight on connection from q th hidden unit to k th output unit: w_{qk}
- weighted input (activation) for q th hidden unit in response to i th input pattern:

$$u_{iq} = \sum_{j=0}^p w_{jq} x_{ij}$$

- weighted input (activation) for k th output unit in response to i th input pattern:

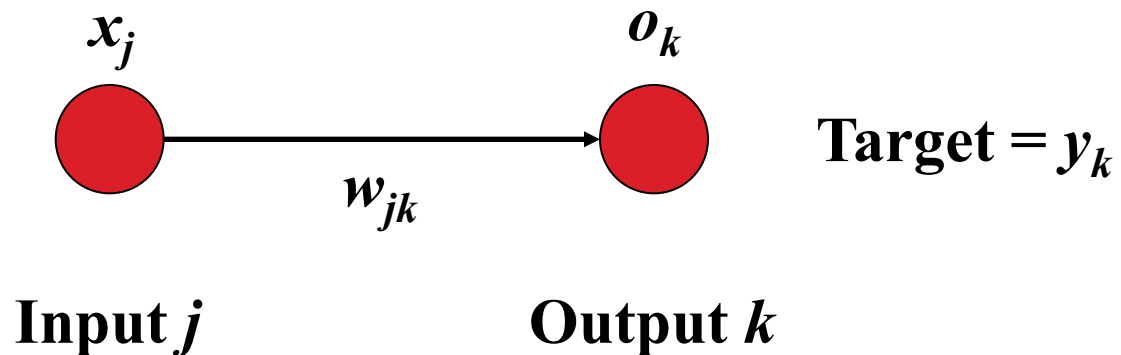
$$z_{ik} = \sum_{q=0}^l w_{qk} h_{iq}$$

- $g(x)$ unit activation function, e.g. sigmoid

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

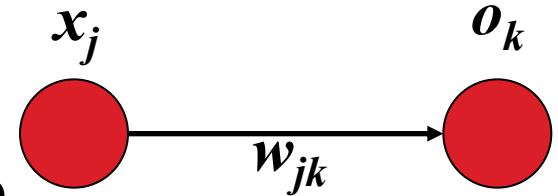
Perceptron Learning Rule (1)

- Adjusting weights to minimise the error at the output of the network
- y_k = target of output k
- o_k = actual output k
- x_j = input j
- w_{jk} = weight between input j and output k



Perceptron Learning Rule (2)

- If $o_k = y_k$
 - No weight change needed
- If $o_k = 1, y_k = 0, y_k - o_k = -1$
 - Weighted input to output k is too large



$$\Delta w_{jk} = -\eta x_j$$

- If $o_k = 0, y_k = 1, y_k - o_k = +1$
 - Weighted input to output k is too small

$$\Delta w_{jk} = +\eta x_j$$

- Gives us:

$$\Delta w_{jk} = (y_k - o_k) \eta x_j$$

Learning: guided by performance

- Measure of performance: sum squared error

$$E = \sum_{i=1}^{n_{tr}} \sum_{k=1}^m E_{ik} = \sum_{i=1}^{n_{tr}} \sum_{k=1}^m [y_{ik} - f_{network,k}(x_i)]^2$$

- y_{ik} is the k th correct output label (value) for training pattern i
- $f_{network,k}(x)$ is the k th element of the overall neural network function (i.e. k th output o_{ik}).

Learning: error rates

$$E = \sum_{i=1}^{n_{tr}} \sum_{k=1}^m E_{ik} = \sum_{i=1}^{n_{tr}} \sum_{k=1}^m [y_{ik} - f_{network,k}(x_i)]^2$$

- Can divide E by the number of training patterns n_{tr} to get average squared error per observation

- Textbook has

$$E = \frac{1}{2} \sum_{i=1}^{n_{tr}} \sum_{k=1}^m E_{ik}$$

- Constant multipliers don't really matter – absorbed in learning rate η during the weight update:

$$w = w - \eta \frac{\partial E}{\partial w} = w + \Delta w$$

Learning: optimisation

- The neural network models the training set best when the error is minimised
- Parameters:
 - weights
 - biases
 - (number of hidden units)
- Assume that each output is equally important
 - Could weight the errors from different outputs

Optimisation Algorithms

- Gradient descent
- 2nd order methods
- Evolutionary algorithms
- Expectation-maximisation algorithm
- Simulated annealing

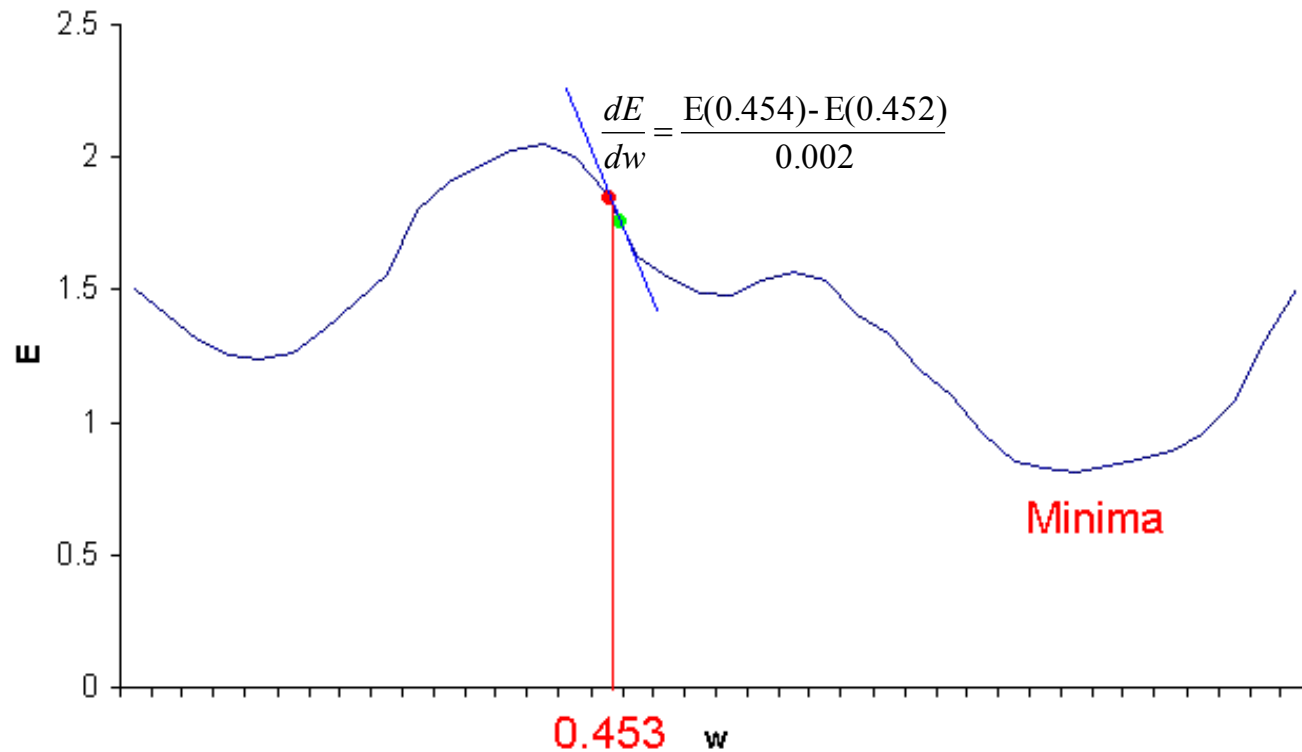
Gradient Descent

- Gradient is often available, so it makes sense to use it (follow the gradient downhill)

Numerical derivation

$$\Delta w \propto -\frac{\partial E}{\partial W}$$

Error surface



Basic gradient descent learning

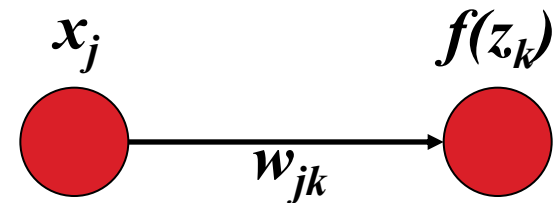
- Minimise the error using the following:
 - Choose a random set of initial weights, e.g. from $U[-0.5,0.5]$ (uniform distribution)
 - Repeat:
 - Run an input training pattern through the network and record the activations of hidden units and output units
 - Calculate E for the current weights.
 - Update every weight via $w = w - \eta \frac{\partial E}{\partial W}$
 - Until stop condition, e.g. happy with performance or nothing much changing
- $\eta > 0$ is the learning rate, typically 0.1

Gradient Descent in Single Layer Neural Networks

$$w_{jk,new} = w_{jk} + \eta \times E_k \times f'(z_k) \times x_j$$

- $w_{jk,new}$ is the updated weight between input j and output k
- w_{jk} is the current weight between input j and output k
- η is the learning rate
- E_k is the error for output k
 $E_k = (y_k - f(z_k))$
- z_k is the weighted input to output k

$$z_k = \sum_{j=0}^n x_j w_{jk}$$



- $f'(z_k)$ is the derivative of the activation function $f(z_k)$
For the sigmoid function $f'(z_k) = f(z_k)(1-f(z_k))$
- x_j is the activation of the input j

Learning in Multi-layer Perceptrons

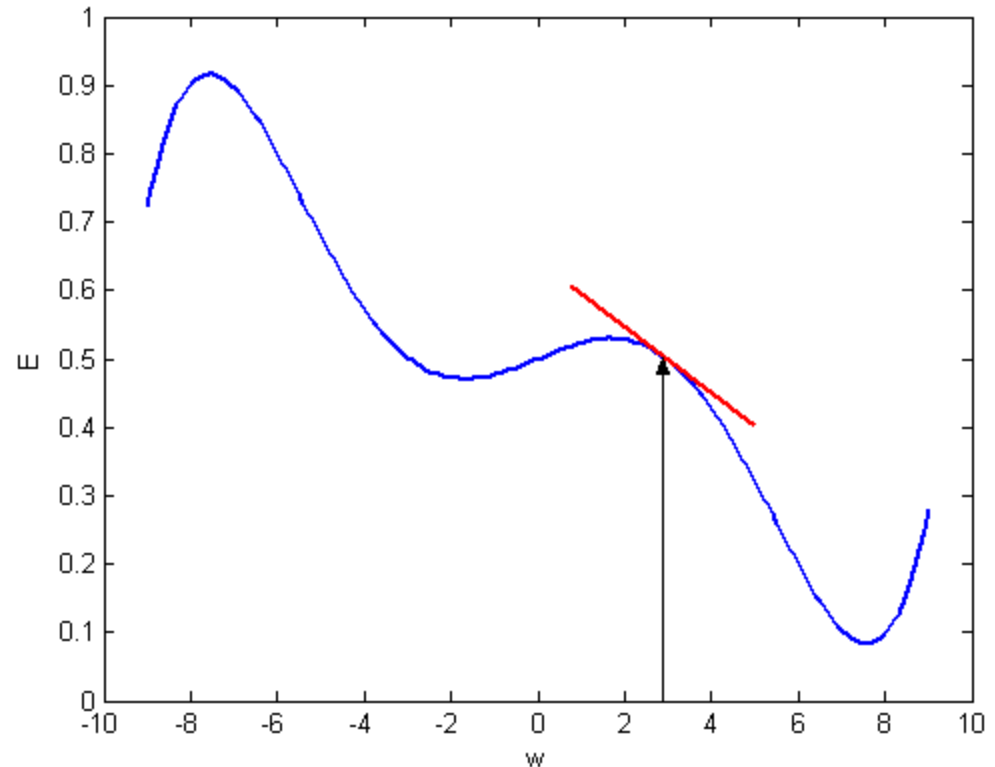
- Multi-layer networks (with a hidden layer) can approximate any function (including those that are not linearly separable)
- Accuracy depends upon
 - how complex the true function mapping input to output is
 - amount of data available (tells us about this function)
 - number of hidden units
 - effectiveness of learning, ie: optimisation of the weights

Backpropagation

- Backpropagation works by gradient descent of the error function in the space of network weights
- For continuous units, error function is differentiable

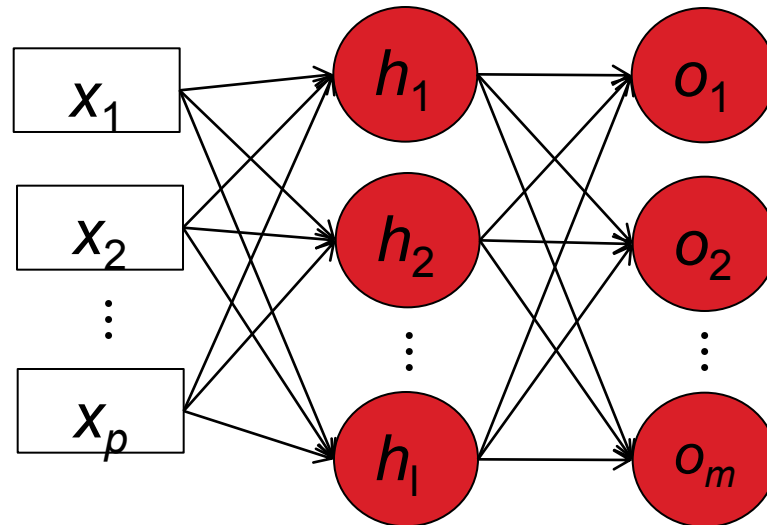
$$E = \sum_{i=1}^{n_{tr}} \sum_{k=1}^m E_{ik}$$

$$\Delta w \propto -\frac{\partial E}{\partial W}$$



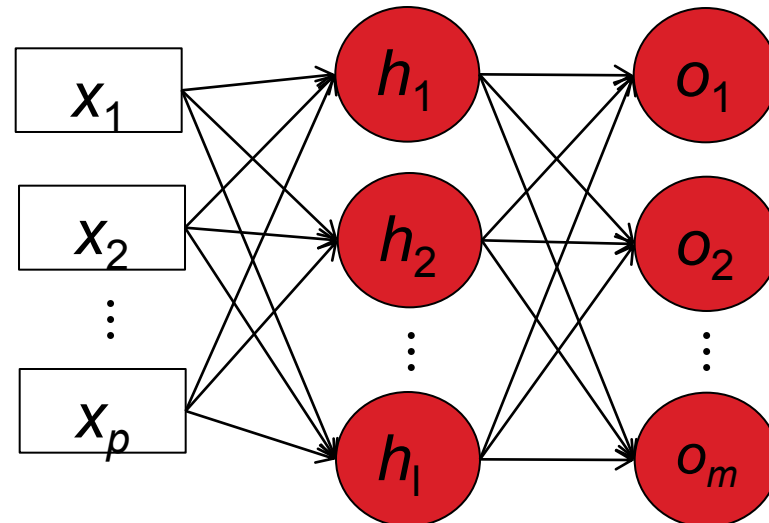
Weight Space (1)

- There are a lot of weights in a neural network
- With a single hidden layer, every input connected to every hidden unit and every hidden unit connected to every output unit + biases on all units, there are $(p+1)l + (l+1)m$ weights
- For example: NETtalk had $p=7 \times 29=203$ inputs, $l=80$ hidden units, $m=26$ output units
- So that is: $204 \times 80 + 81 \times 26 = 18,426$ weights
- We are trying to search for a global minimum of E over that space – barely imaginable



Weight Space (2)

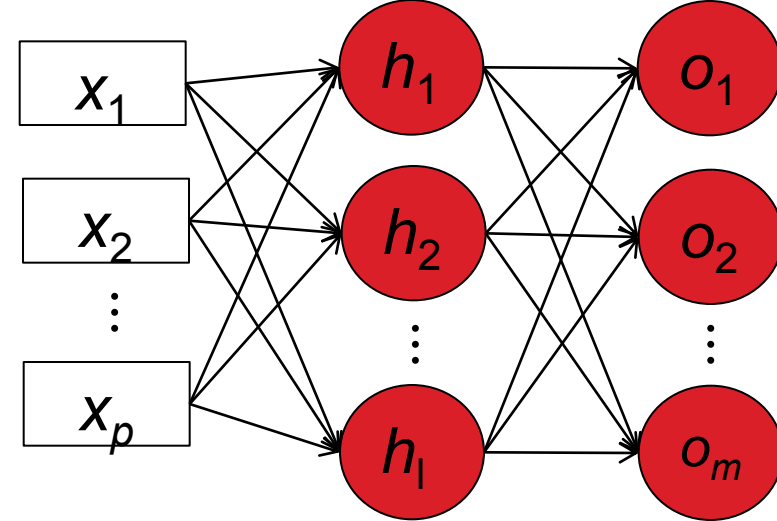
- From one perspective, you can never get enough data to accurately determine so many weight values
- But many near-optimal solutions exist which will do quite well in practice
- These can be found based on limited data, and with simple optimisation methods like gradient descent
- It is useful to use a smaller network where possible (fewer hidden units):
 - Easier to train
 - Likely better generalisation



Backpropagation strategy

- Networks with hidden nodes can be trained – non-linearly separable problems can be learned
- The overall strategy is:
 - Calculations based on errors at the output are propagated backwards through the network
 - Each hidden node receives some “blame” based on its contribution to the output error
 - As a result, hidden activations come to represent higher order features of the input, useful for the learning task

1. Feed forward
2. Error calculation
3. Weight modification



Feed forward

$$g(u) = \frac{1}{1 + e^{-u}}$$

$$u_{iq} = \sum_{j=0}^p w_{jq} x_{ij}$$

$$h_{iq} = g(u_{iq})$$

$$z_{ik} = \sum_{q=0}^l w_{qk} h_{iq}$$

$$o_{ik} = g(z_{ik})$$

Error calculation

$$E = \sum_{i=1}^{n_{tr}} \sum_{k=1}^m E_{ik}$$

$$E_{ik} = [y_{ik} - o_{ik}]^2$$

Weight modification

$$w = w_{old} + \Delta w$$

$$\Delta w = -\eta \frac{\partial E}{\partial w}$$

Backpropagation algorithm for learning in multilayer networks

function BACK-PROP-LEARNING(*examples*, *network*) **returns** a neural network

inputs: *examples*, a set of examples, each with input vector \mathbf{x} and output vector \mathbf{y}

network, a multilayer network with L layers, weights $W_{j,i}$, activation function g

repeat

for each e in *examples* **do**

for each node j in the input layer **do** $a_j \leftarrow x_j[e]$ **Set the input activations**

for $l=2$ to L **do** $in_i \leftarrow \sum_j W_{j,i} a_j$ **For each layer, feedforward the activations**

$$a_i \leftarrow g(in_i)$$

for each node i in the output layer **do**

$\Delta_i \leftarrow g'(in_i) \times (y_i[e] - a_i)$ **For each output, determine the error**

for $l=L-1$ to 1 **do**

for each node j in layer l **do**

$$\Delta_j \leftarrow g'(in_j) \sum_i W_{j,i} \Delta_i$$

for each node i in layer $l+1$ **do**

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

For each layer, determine the 'blame' attributed to each unit and work out the new weights

Repeat until the performance has reached the desired level, or until the max training time

until some stopping criterion is satisfied

return NEURAL-NET-HYPOTHESIS(*network*)

Backpropagation Derivation (1)

Assume logistic unit activation function $g()$

Weight update rule for hidden-output weights w_{bc} (includes output unit bias weights), $b \in 0, \dots, l$, $c \in 1, \dots, m$:

$$\Delta w_{bc} = -\eta \frac{\partial E}{\partial w_{bc}} = -\eta \frac{\partial}{\partial w_{bc}} \sum_{i=1}^{n_{tr}} \sum_{k=1}^m E_{ik} = -\eta \sum_{i=1}^{n_{tr}} \frac{\partial E_{ic}}{\partial w_{bc}}$$

$$\frac{\partial E_{ic}}{\partial w_{bc}} = \frac{\partial [y_{ic} - o_{ic}]^2}{\partial w_{bc}} = -2[y_{ic} - o_{ic}] \frac{\partial o_{ic}}{\partial w_{bc}}.$$

$$o_{ic} = g(z_{ic}) = g\left(\sum_{q=0}^l w_{qc} h_{iq}\right).$$

$$g(z_{ic}) = \frac{1}{1 + e^{-z_{ic}}}, \text{ so } \frac{\partial g(z_{ic})}{\partial w_{bc}} = \frac{e^{-z_{ic}}}{(1 + e^{-z_{ic}})^2} \frac{\partial z_{ic}}{\partial w_{bc}} = e^{-z_{ic}} o_{ic}^2 h_{ib}$$

$$\text{So } \Delta w_{bc} = 2\eta \sum_{i=1}^{n_{tr}} [y_{ic} - o_{ic}] o_{ic}^2 e^{-z_{ic}} h_{ib}$$

If $b = 0$ (bias weight on an output unit), $h_{i0} = 1$ in above equation.

Backpropagation Derivation (2)

Weight update rule for input-hidden layer weight w_{ab} (includes hidden unit bias weights), $a \in 0, \dots, p$, $b \in 1, \dots, l$:

$$\Delta w_{ab} = -\eta \frac{\partial E}{\partial w_{ab}} = -\eta \frac{\partial}{\partial w_{ab}} \sum_{i=1}^{n_{tr}} \sum_{k=1}^m E_{ik} = -\eta \sum_{i=1}^{n_{tr}} \sum_{k=1}^m \frac{\partial E_{ik}}{\partial w_{ab}}$$

$$\frac{\partial E_{ik}}{\partial w_{ab}} = \frac{\partial [y_{ik} - o_{ik}]^2}{\partial w_{ab}} = -2[y_{ik} - o_{ik}] \frac{\partial o_{ik}}{\partial w_{bc}}$$

$$o_{ik} = g(z_{ik}) = g\left(\sum_{q=0}^l w_{qk} h_{iq}\right).$$

$$g(z_{ik}) = \frac{1}{1 + e^{-z_{ik}}}, \text{ so } \frac{\partial g(z_{ik})}{\partial w_{ab}} = \frac{e^{-z_{ik}}}{(1 + e^{-z_{ik}})^2} \frac{\partial z_{ik}}{\partial w_{ab}} = e^{-z_{ik}} o_{ik}^2 w_{bk} \frac{\partial h_{ib}}{\partial w_{ab}}$$

$$\frac{\partial h_{ib}}{\partial w_{ab}} = \frac{\partial g(u_{ib})}{\partial w_{ab}} = e^{-u_{ib}} h_{ib}^2 \frac{\partial u_{ib}}{\partial w_{ab}}$$

$$\frac{\partial u_{ib}}{\partial w_{ab}} = \frac{\partial}{\partial w_{ab}} \sum_{j=0}^p w_{jb} x_{ij} = x_{ia}$$

$$\text{So } \Delta w_{ab} = 2\eta \sum_{i=1}^{n_{tr}} \sum_{k=1}^m [y_{ik} - o_{ik}] e^{-z_{ik}} o_{ik}^2 w_{bk} e^{-u_{ib}} h_{ib}^2 x_{ia}$$

Backpropagation Implementation

- Noting the following update rules for the hidden-output and input-hidden weights, it becomes apparent what needs to be stored on the feedforward phase (producing output) so that backpropagation learning can be performed

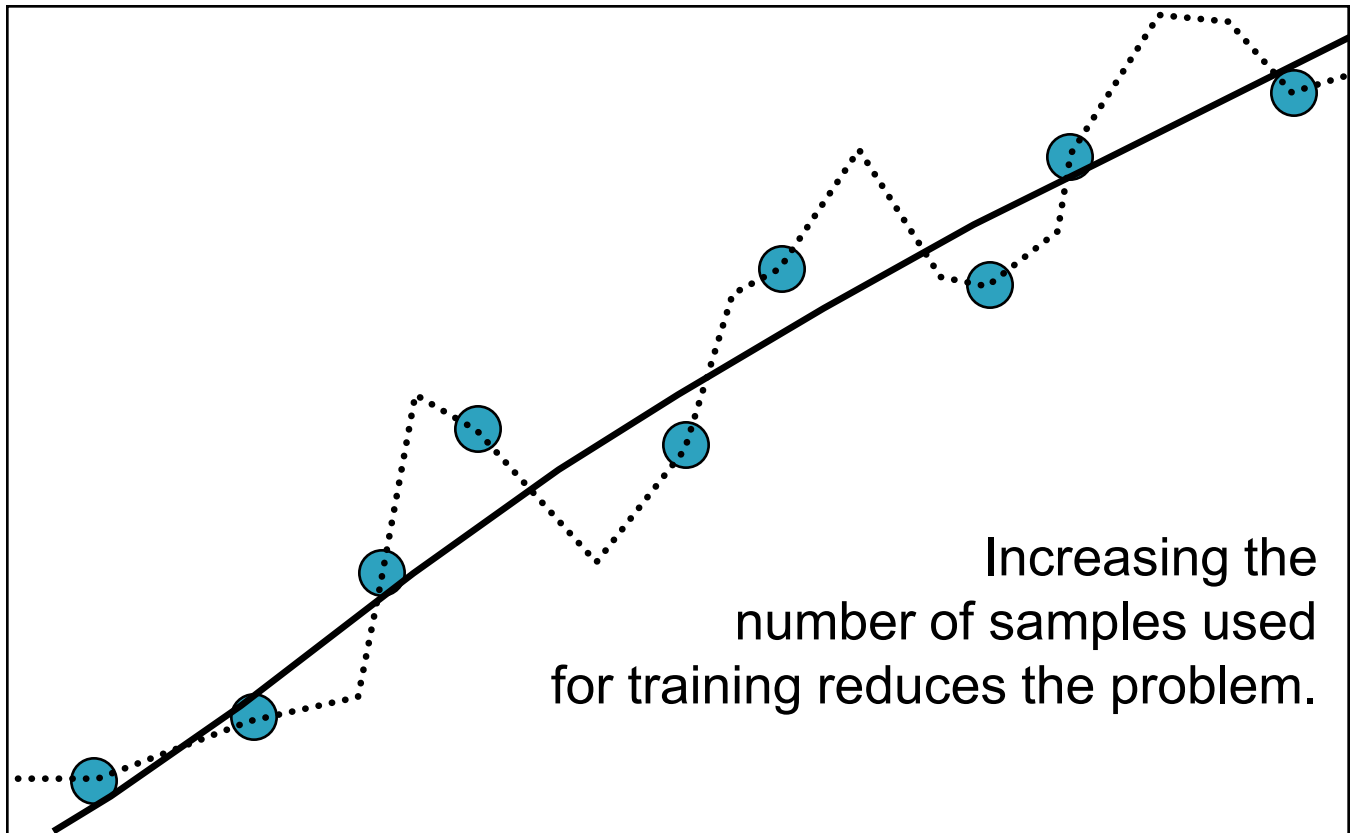
$$\begin{aligned}\Delta w_{bc} &= 2\eta \sum_{i=1}^{n_{tr}} [y_{ic} - o_{ic}] e^{-z_{ic}} o_{ic}^2 h_{ib} \\ \Delta w_{ab} &= 2\eta \sum_{i=1}^{n_{tr}} \sum_{k=1}^m [y_{ik} - o_{ik}] e^{-z_{ik}} o_{ik}^2 w_{bk} e^{-u_{ib}} h_{ib}^2 x_{ia} \\ &= 2\eta \sum_{i=1}^{n_{tr}} e^{-u_{ib}} h_{ib}^2 x_{ia} \left(\sum_{k=1}^m [y_{ik} - o_{ik}] e^{-z_{ik}} o_{ik}^2 w_{bk} \right)\end{aligned}$$

- So, in response to each training input, we need to store all the network hidden and output values and their activations (weighted input)

Learning and Generalisation

- Want the network to learn the training set and classify these examples correctly
- Also want the network to be able to generalise to a test set
- There can be a trade-off between learning and generalisation, especially when there are errors or noise in the training set

Generalization or over fitting?



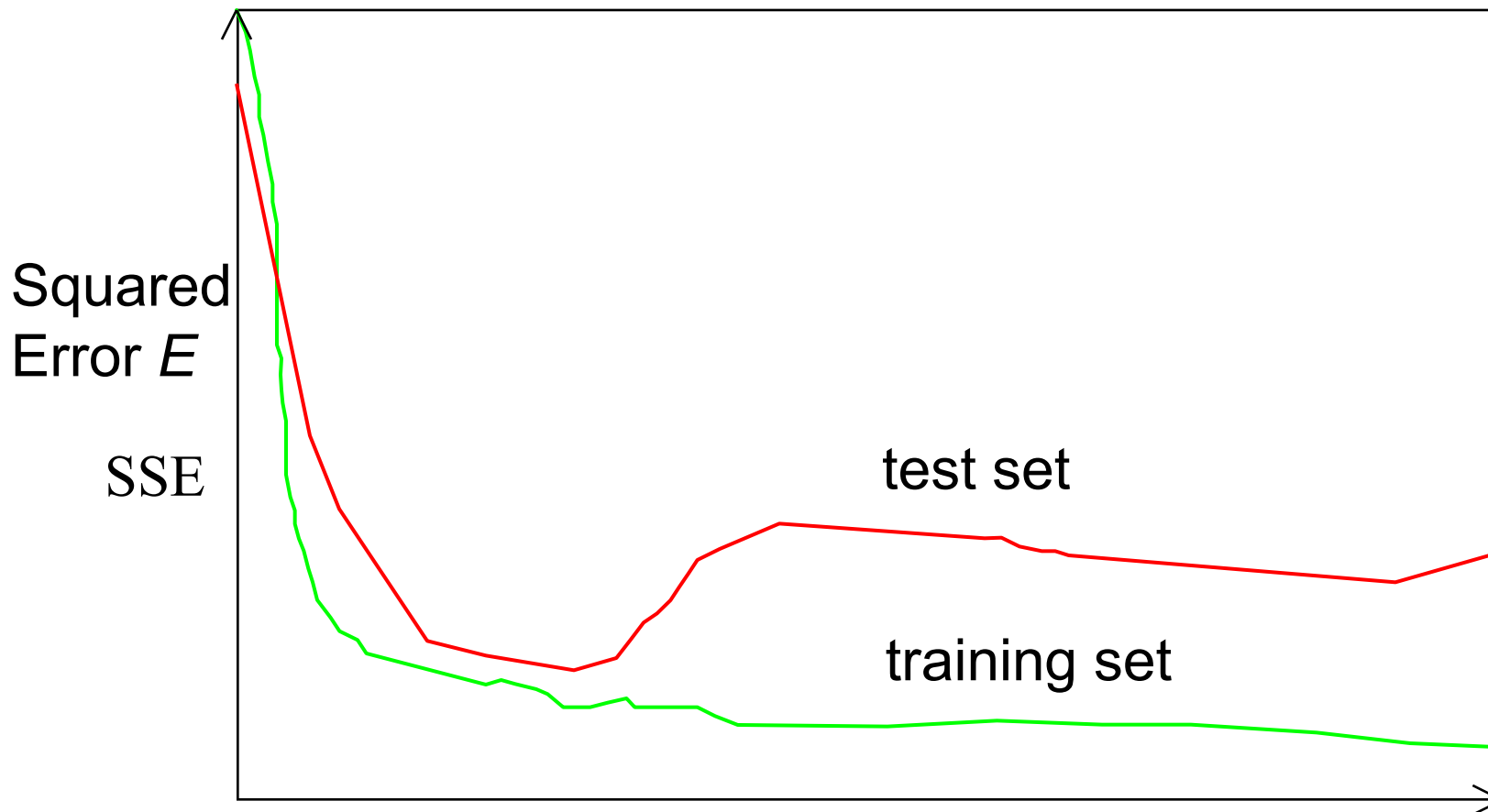
Generalisation:

- Minimising error on the training set is a strategy to obtain good performance on new data
- It doesn't always work
- Possible reasons:
 - Over-fitting
 - Under-fitting

Generalisation: Over-fitting

- Model is a very good fit to the training data
- May have modelled some useful structure in the data
- Has also modelled many irrelevant aspects
- Common with overly-complex models
- Solution: early stopping

What often happens:



epochs = iterations of learning algorithm

- This NN may be too complex for the data/problem and so prone to overfitting
- If we stop training early, performance could be ok

Controlling generalization by using validation data

- Dataset is divided into a training set, test set and a validation set
- Training set is used for optimizing weights
- Validation set is monitored during (or after) training for optimizing model parameters (when to stop, #nodes/weights, learning rate etc)
- Test set is used to assess how well the model performs

Generalisation: Under-fitting

- Model is not a good fit to the training data (size of E is a good indication)
- Model may be too simple
- You may not have enough data
- The data may be so noisy that you can never predict the output very accurately
- Solution: structure of network, quality and quantity of data

Parameter Values and Other Issues

- Weight initialization
 - Typically small random values with mean 0
- Learning methods
 - Batch or online updates
- Learning rate
 - $0 < \eta \leq 1$, typically 0.1
- Stopping criterion
- Number of layers
- Number of nodes in hidden layers
- Number of examples

Learning methods

- Batch learning
 - Initialise the weights
 - Repeat as necessary:
 - Process all training data
 - Update weights
- Online learning
 - Initialise the weights
 - Repeat as necessary:
 - Process one training example
 - Update weights

Batch learning

- Requires us to feed in all n_{tr} training patterns and record hidden and output unit activations and outputs
- Batch NN learning tends to follow: large-scale, medium-scale, then small-scale optimisation
 - If misled early on, might never recover
 - Rare patterns might be missed

Online learning (1)

- Sometimes we have a lot of data, e.g. >5000 bitmap patterns for Assignment 2
- If doing e.g. 10,000 epochs (iterations) of learning, this results in a lot of calculations
- Alternative: online mode:
 - At each iteration, choose 1 training pattern at random, and update based on this alone
- Is implemented in Assignment 2 supplied code

Online learning (2)

- Advantages:
 - feedforward phase is n_{tr} times quicker
 - Can escape local minima due to constant shifting of targets
 - Eventually sees all training patterns and does the right thing “on average”
 - Can cope with non-stationary targets, ie: if the input-output function is changing over time, this method can react to changing training data

Online learning (3)

- Disadvantages:
 - May have trouble converging to a solution: current solution is not optimal for any single training pattern
 - In practice, seems to perform quite well, but probably needs more iterations than batch learning

Building a Neural Network

- Specify the inputs and outputs of the problem
- Choose the simplest structure that will work
- Find appropriate connection weights
- Check the network works on the training data, and test generalisation on test data
- If performance is not good enough, improve one of these features

Encoding v output classes

- Simplest option: v outputs: for each: 1 means it is of that class, 0 means it isn't
- Treats each class equally
- The network is never completely sure which output class it is seeing, and can produce a number of output values in the range $(0,1)$
- Choose the output with highest value as the prediction
- Could interpret output values as probabilities by using a normalising constant

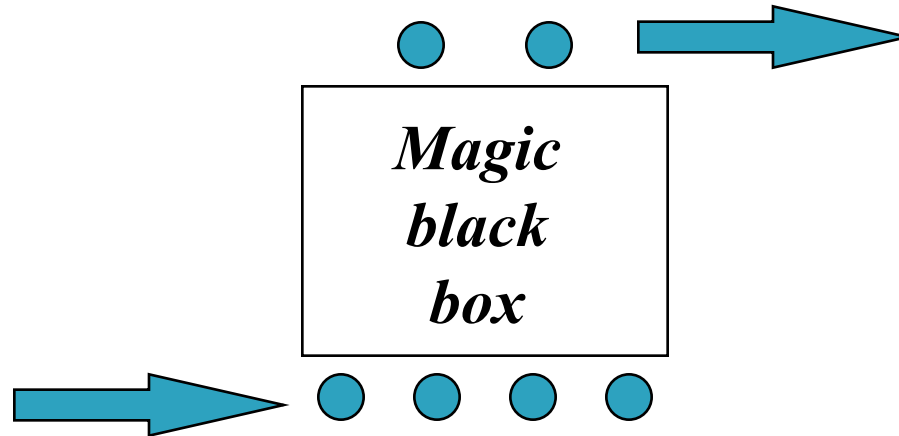
The network

- Choose a network topology
- Determine the number of hidden units, if any
- Create network
 - NN1(int nInput, int nOutput, int seed)
 - NN1(String filename)
- Learning
 - double train(double[] x, double[] d, double eta)
- Evaluate the network configuration by running tests
 - double[] feedforward(double[] x)

Evaluating the network

- Record generalization performance on different test sets
 - Provides no model-specific explanation
- Inspecting internal activation
 - Provides knowledge on the effect of different inputs on the model
- Studying network weights
 - Provides knowledge on decisions made in the model

Testing different inputs



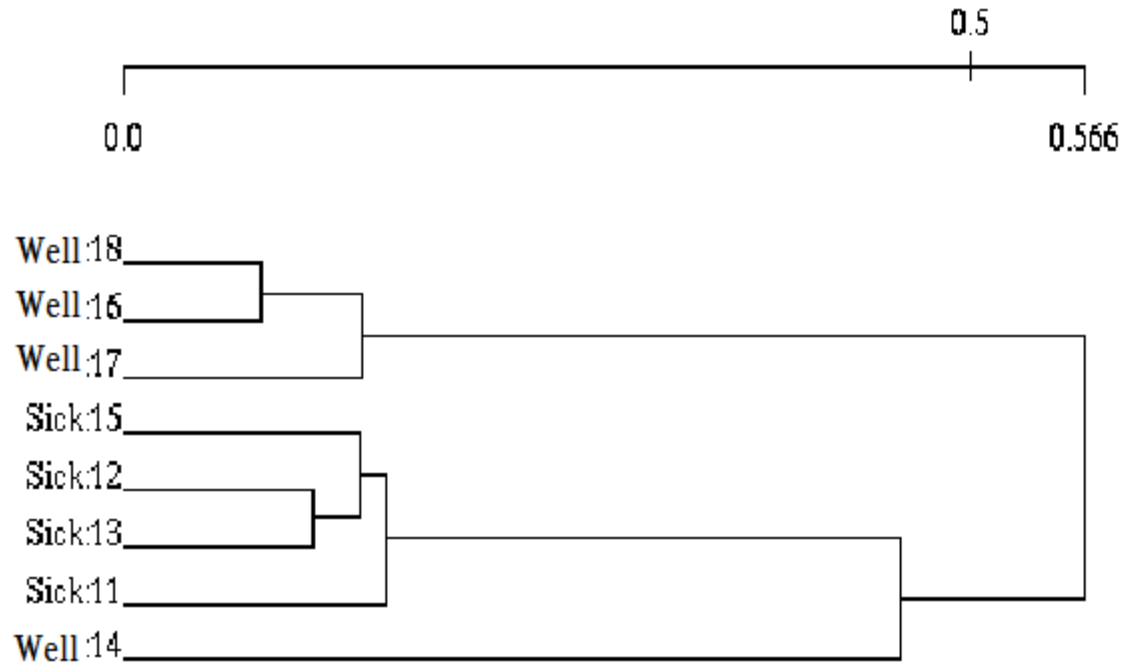
Network 1 (seed 1): SSE=0.001, 89% correct on test set

Network 2 (seed 2): SSE=0.021, 91% correct on test set

Network 3 (seed 3): SSE=0.005, 77% correct on test set

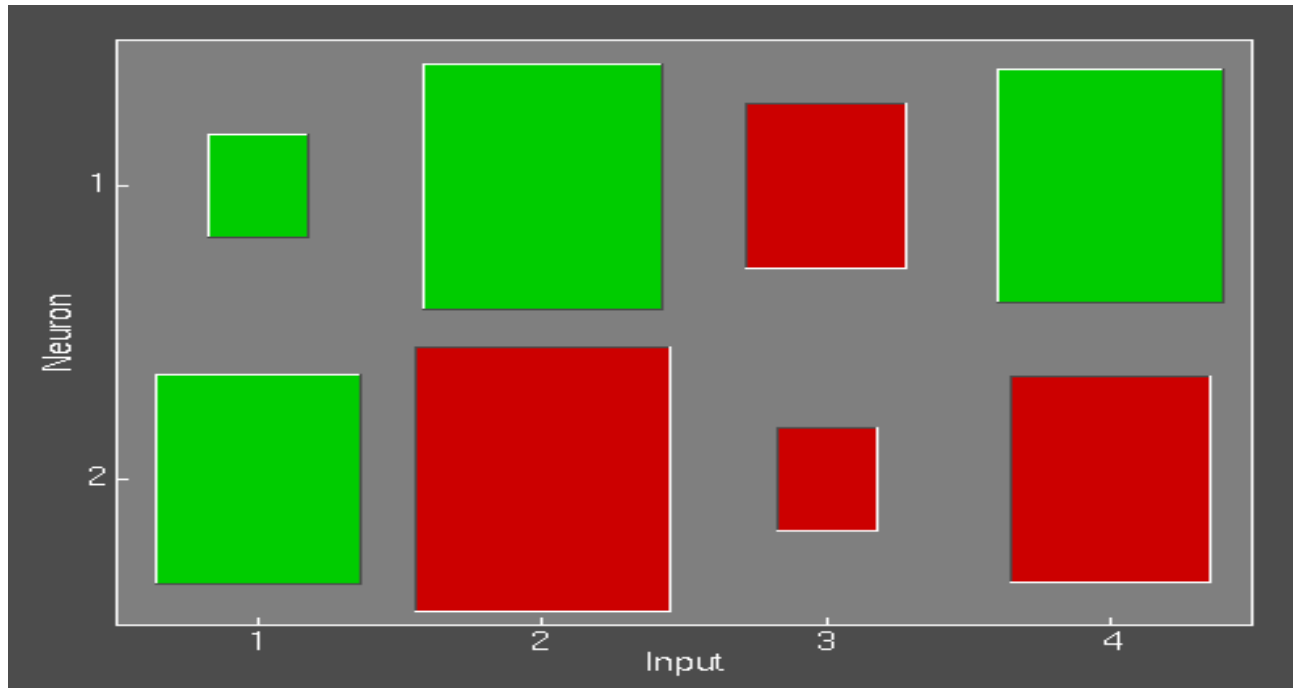
Network 1 has trouble with p_1 and p_2 . What is special about them?

Inspecting internal activation



Visualize (spatial) similarity between internal activations

Studying weights



Hinton diagram visualizes information how weights are used, which units are relevant

hintonw in Matlab

Summary

- Units
- Decision Boundaries
- Logic Functions
- Single and Multi Layer Networks
- Learning
 - Gradient Descent
 - Backpropagation
- Generalisation
- Learning methods
 - Batch
 - Online
- Building a Neural Network