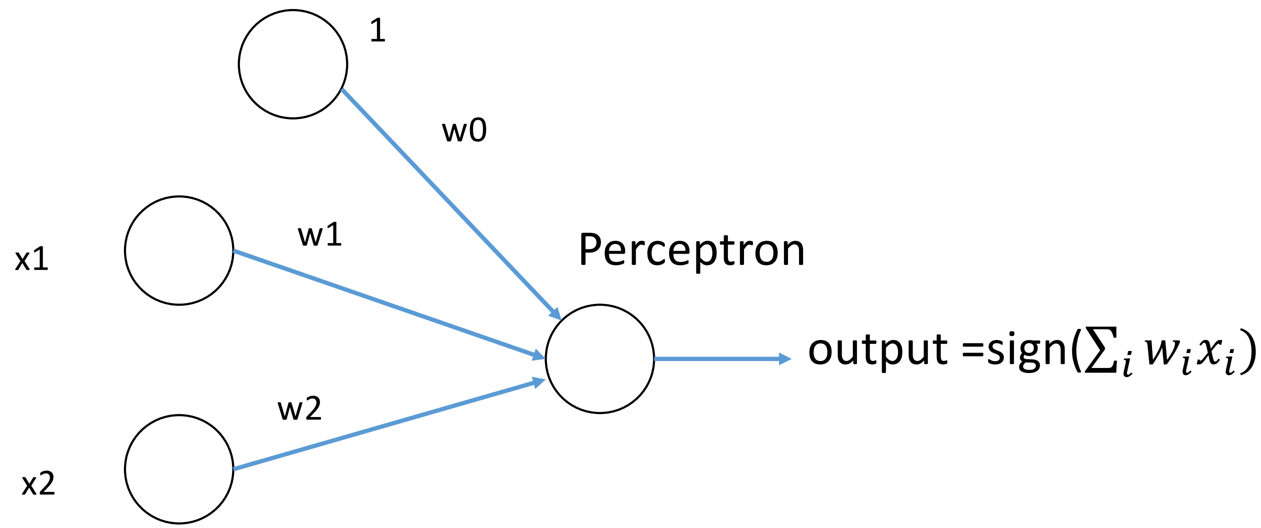
Neural Net

Extending Perceptron

- Perceptron only works with linearly separable data.
- How do we make it learn more complicated, non-linear datasets?
- Change the output function from step to a non-linear one.
- Strength in unity
- Combine multiple perceptron units

What is a NN?

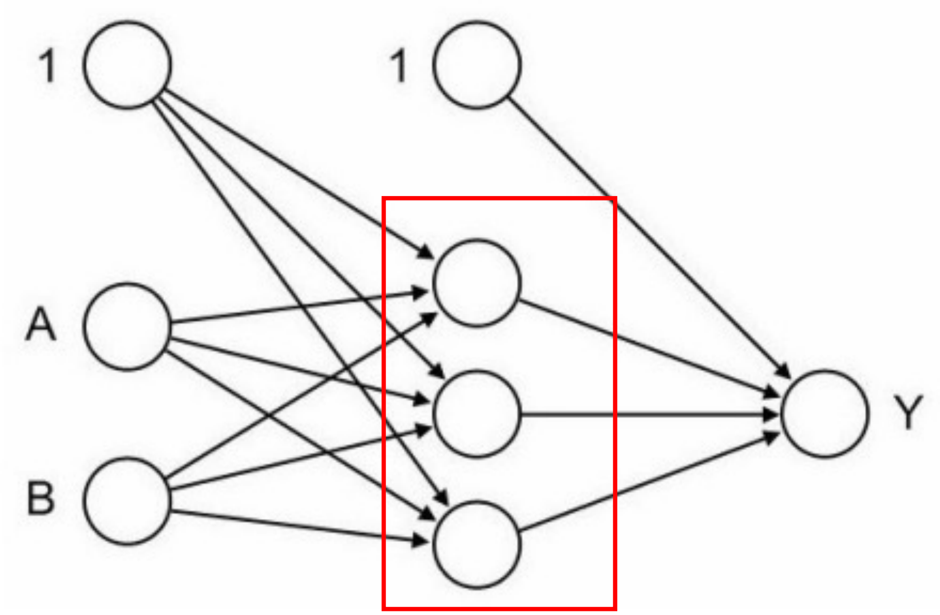
- We saw earlier that a single perceptron can only separate linear data



What is a NN?

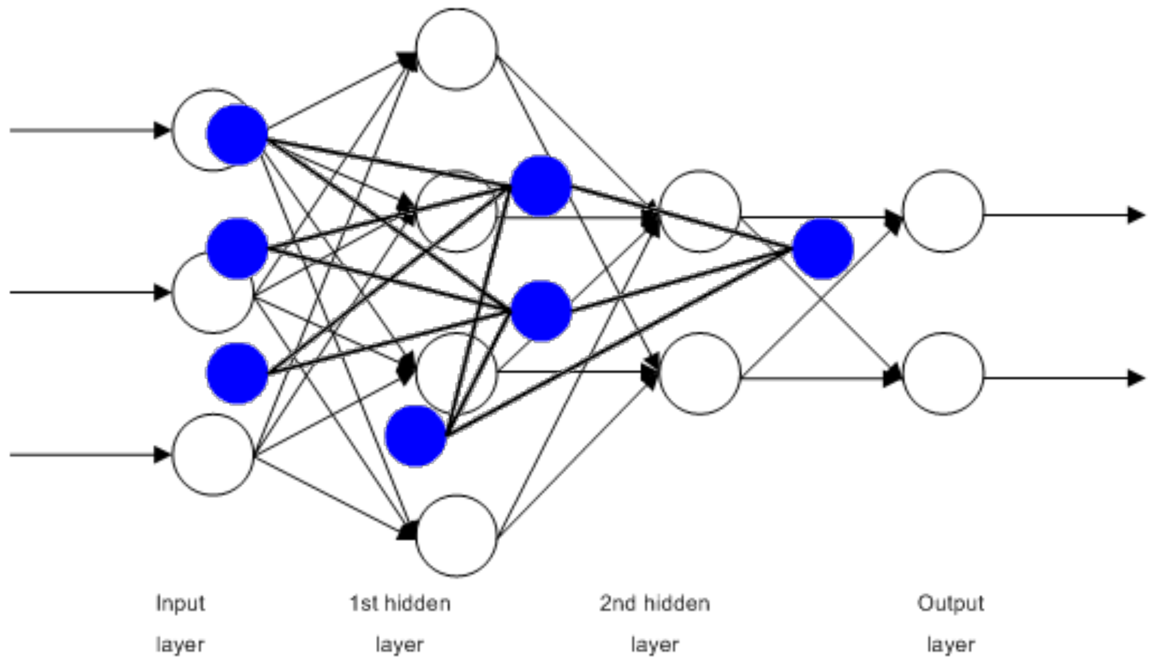
- What would happen if we create a network of perceptrons.
- Initial input is fed to a set of nodes in the intermediate layer, known as hidden layer.
- Output from hidden layer can be sent to another hidden layer or output

Hidden Layers that
perform transformation
of input signals to outputs



Neural Network

- A set of perceptrons joined together in multiple layers.
- The output can represent highly complex and non-linear functions.
- By constructing this network, we use a combination of simple perceptrons to build a powerful classifier.



Properties of NN

- A large number of very simple neuron-like processing elements.
- A large number of weighted connections between the elements.
- Highly parallel, distributed control.
- An emphasis on learning internal representations automatically.

ANN as universal approximator

- We can approximate any Boolean function and almost all continuous functions with a multi-layer perceptron.
- Let's check the XOR function:
  Remember:
  x1 XOR x2 = (x1 AND NOT x2) OR (NOT x1 AND x2)

ANN as universal approximator

- Let's check the XOR function:
  Remember:
  x1 XOR x2 = (x1 AND NOT x2)
  OR (NOT x1 AND x2)
- In the diagram, step activation function is used everywhere.

$$x_1 \text{ XOR } x_2 = (x_1 \text{ AND } \sim x_2) \text{ OR } (\sim x_1 \text{ AND } x_2)$$

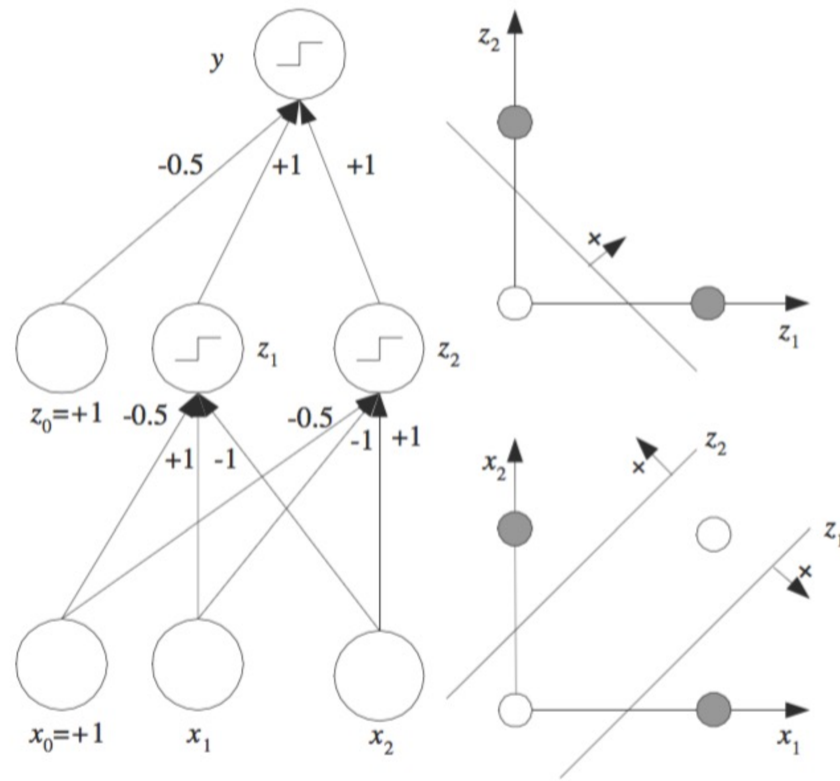
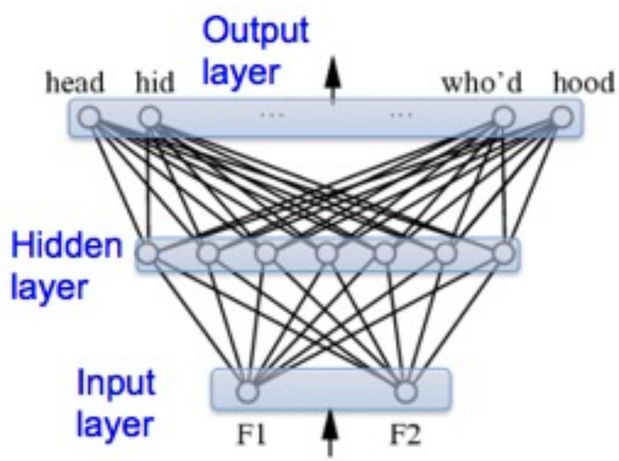


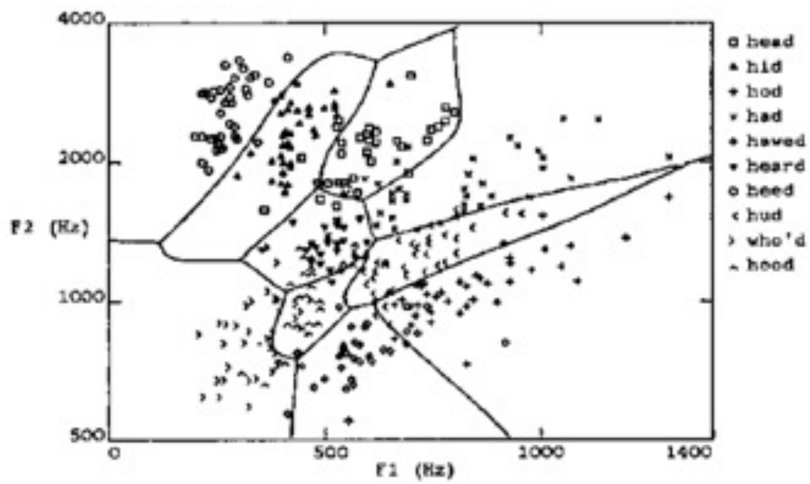
Figure 11.7 of Intro to ML textbook

Applications of NN

Neural Network trained to distinguish vowel sounds using 2 formants (features)



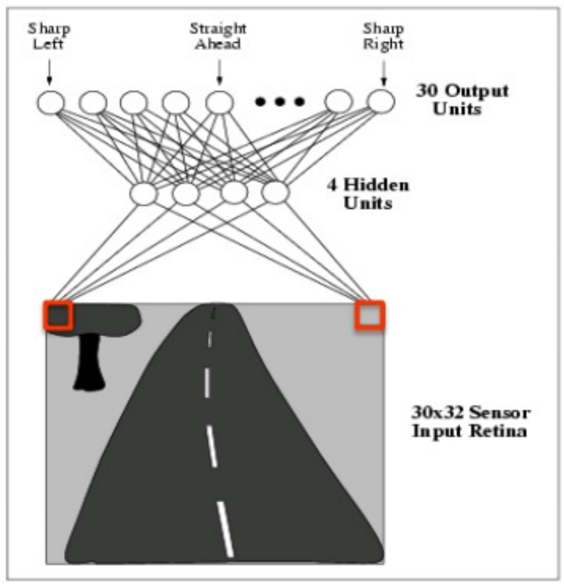
Two layers of logistic units



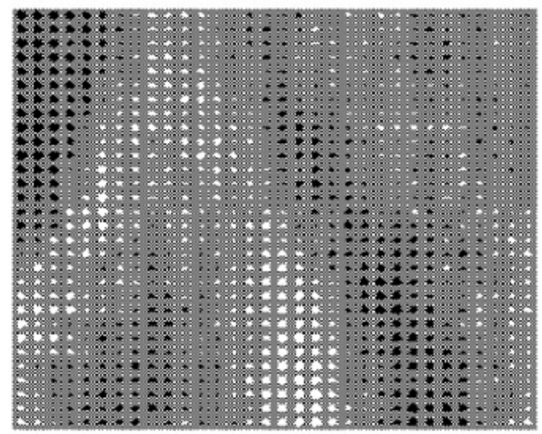
Highly non-linear decision surface

Applications of NN

Neural Network trained to drive a car!



Weights to output units from the hidden unit

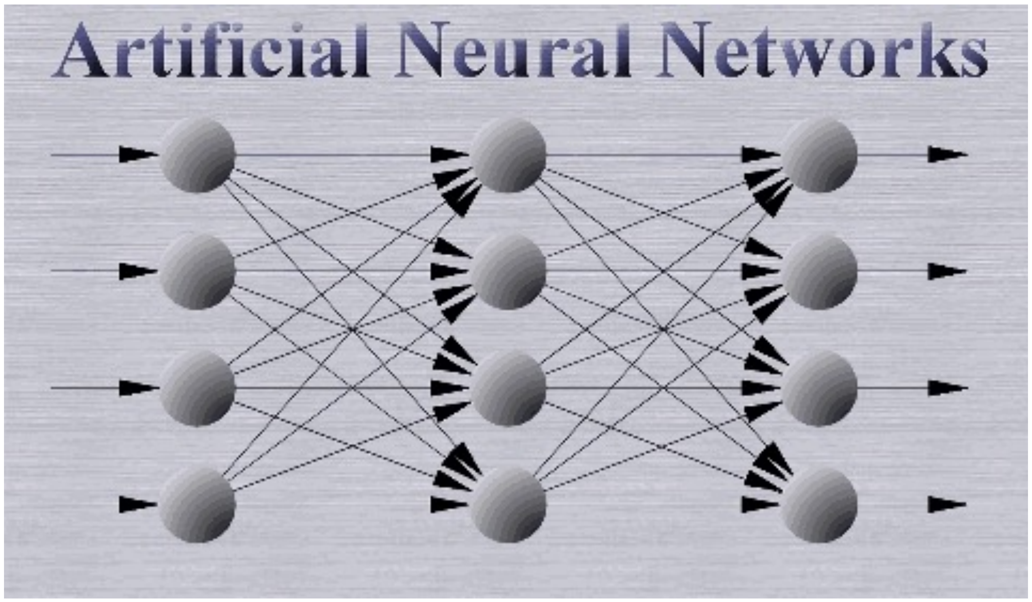


Weights of each pixel for one hidden unit

Training a NN

- All this sounds great, but..
- How do I train this complex network?
- Let's see.

Artificial Neural Network (ANN)

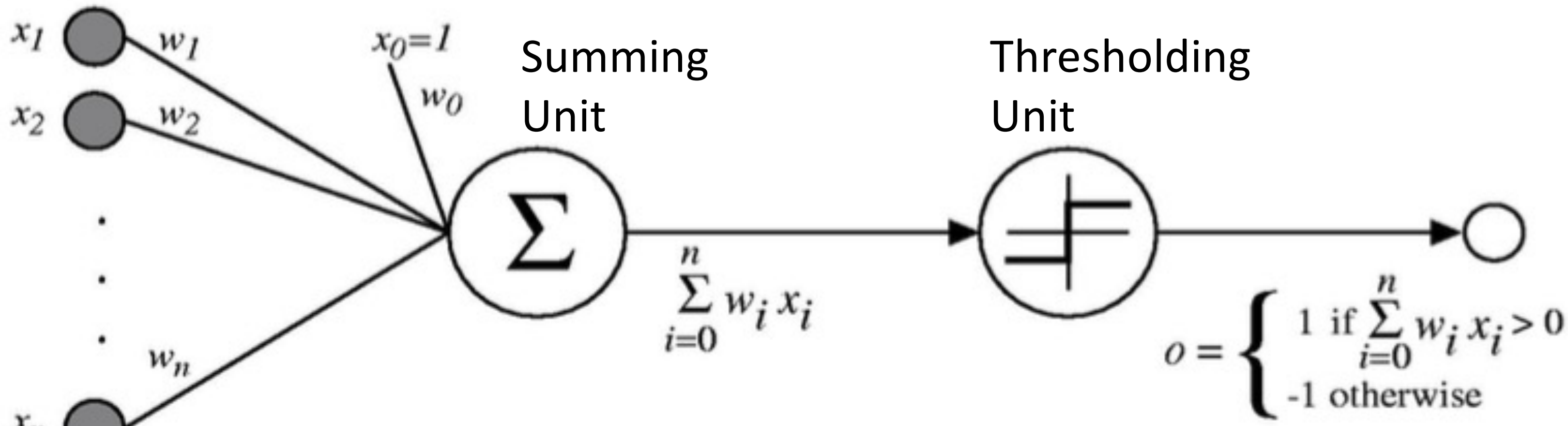


- A group of perceptrons joined together to achieve powerful results.
- Multi-layered processing.
- Each layer and unit has to be trained.
- Once trained, it can achieve great results.

Perceptron

- Let's revise to make sure we still remember the perceptron.
- It has only one decision unit.

Perceptron



Perceptron Training Rule

$$w_i \leftarrow w_i + \Delta w_i$$

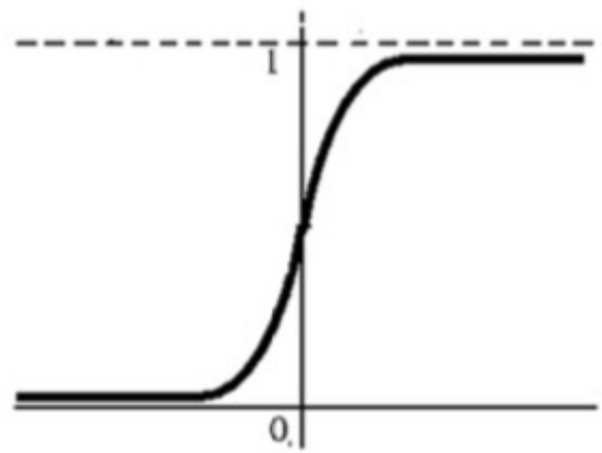
where

$$\Delta w_i = \eta(t - o)x_i$$

How to build a Neural Net

- Remember for a perceptron, there was a summing unit and a linear decision or activation unit. It's called threshold function
- The linear unit works for linearly separable data, but to make it work with other data, we need a non-linear activation unit.
- Sigmoid activation function:

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



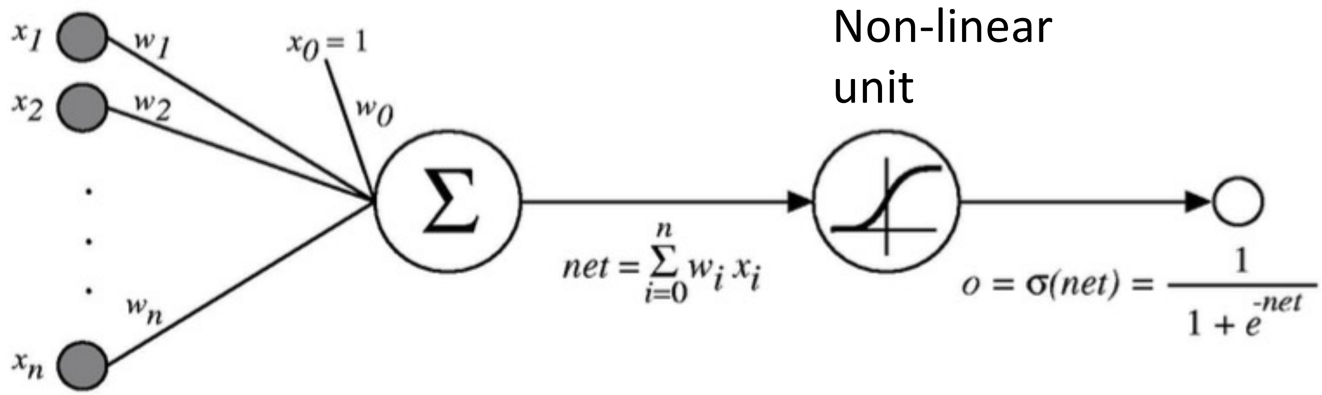
What happens when
x -> -∞ and x -> ∞ ?

How to build a Neural Net

- Remember for a perceptron, there was a summing unit and a linear decision or activation unit.
- The linear unit works for linearly separable data, but to make it work with other data, we need a non-linear activation unit.
- In NN, we will create units that have a non-linear unit applied to the output of the summing unit.

Sigmoid Unit

Sigmoid activation function is applied to net



$\sigma(x)$ is the sigmoid function

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

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$ Really useful result

Derivative of sigmoid unit

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

$$\frac{d\sigma(x)}{dx} = -\frac{1}{(1 + e^{-x})^2}(-e^{-x})$$

$$\frac{d\sigma(x)}{dx} = \frac{1}{(1 + e^{-x})}\frac{e^{-x}}{(1 + e^{-x})}$$

$$\frac{d\sigma(x)}{dx} = \frac{1}{(1 + e^{-x})}(1 - \frac{1}{(1 + e^{-x})})$$

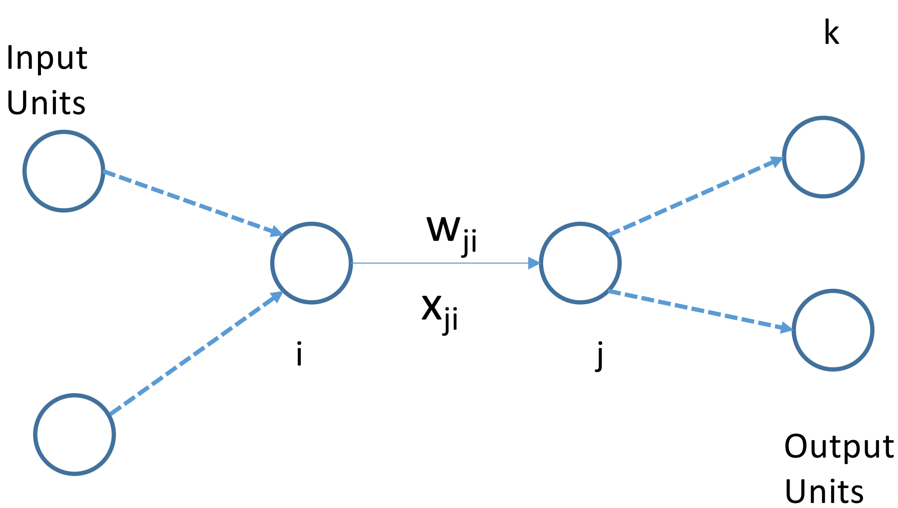
$$\boxed{\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))}$$

We can derive gradient descent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units $\rightarrow$ Backpropagation

Backpropagation Algorithm

- Consider the network shown on right and two intermediate nodes i and j



- Error for example *d* is:

$$E_d(w) = \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

for all output units find squared error and sum

- Weight delta (from last class):

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

How does w_{ji} affect E_d ?

Backpropagation Algorithm

- Let's try to compute $\frac{\partial E_d}{\partial w_{ji}}$

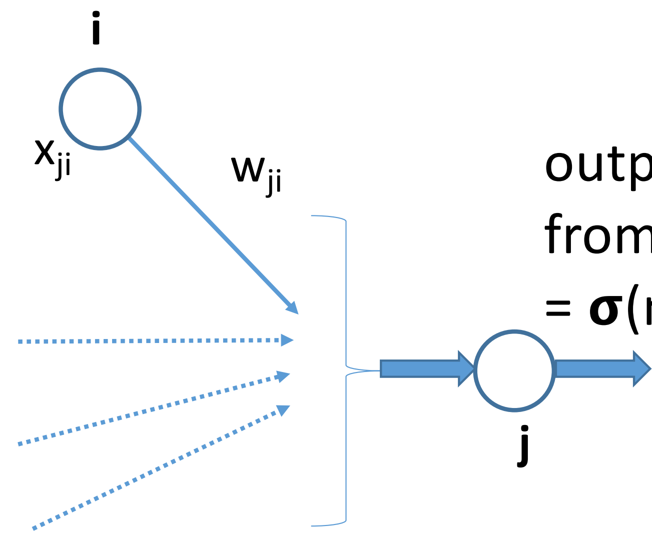
How does w_{ji} affect E_d

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$

- $\frac{\partial net_j}{\partial w_{ji}}$ is easy to compute:

$$\frac{\partial net_j}{\partial w_{ji}} = x_{ji}$$

- What about $\frac{\partial E_d}{\partial net_j}$?



output from j = o_j = $\boldsymbol{\sigma}(net_j)$

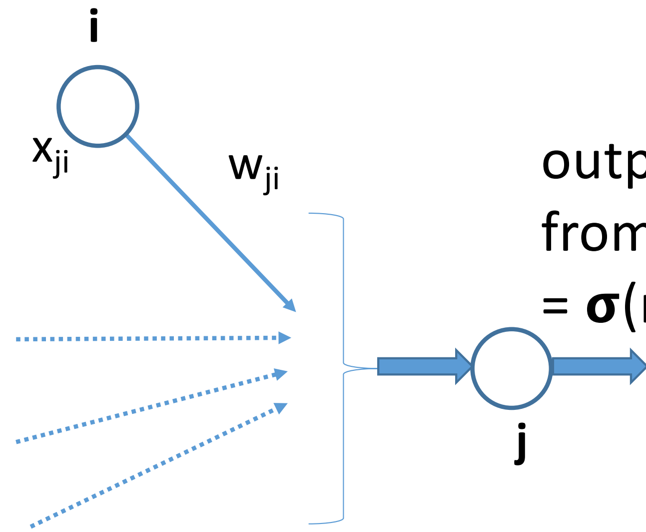
net coming into j = net_j = $\sum_i w_{ji} x_{ji}$

Backpropagation Algorithm

Two cases:

- j is an output unit
- j is a hidden unit

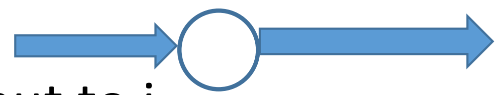
Value of $\frac{\partial E_d}{\partial net_j}$ will be different for the two above cases.



Backpropagation Algorithm

Case I: j is an output unit

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j}$$



We know that error for data *d* is:

$$E_d(w) = \frac{1}{2}\sum_{k \in outputs}(t_k - o_k)^2$$

Differentiating wrt o_j :

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}[\frac{1}{2}\sum_{k \in outputs}(t_k - o_k)^2]$$

$$= \frac{\partial}{\partial o_j}[\frac{1}{2}\left(t_j - o_j\right)^2]$$

$$= -\left(t_j - o_j\ \right)$$

Second term is famous sigmoid derivative:

$$\frac{\partial o_j}{\partial net_j} = o_j(1 - o_j)$$

Backpropagation Algorithm

Case I: j is an output unit:

Putting it all together:

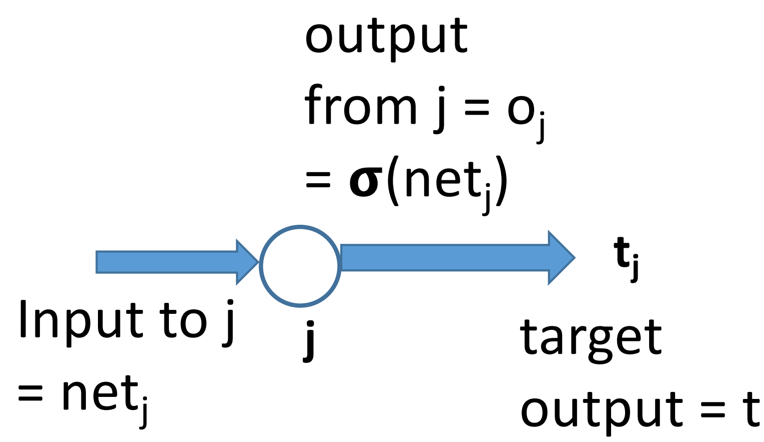
$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)o_j(1 - o_j) = -\delta_j$$

Let's plug this back in Δw_{ji}:

$$\boxed{\begin{aligned}\Delta w_{ji} &= -\eta \frac{\partial E_d}{\partial w_{ji}} \\ &= \eta(t_j - o_j)o_j(1 - o_j)x_{ji}\end{aligned}}$$

If we call $(t_j - o_j)o_j(1 - o_j) = \delta_j$

$$\boxed{\Delta w_{ji} = \eta \delta_j x_{ji}}$$



We will call the partial of the error wrt net for any unit j as follows:

$$\frac{\partial E_d}{\partial net_j} = -\delta_j$$

Backpropagation Algorithm

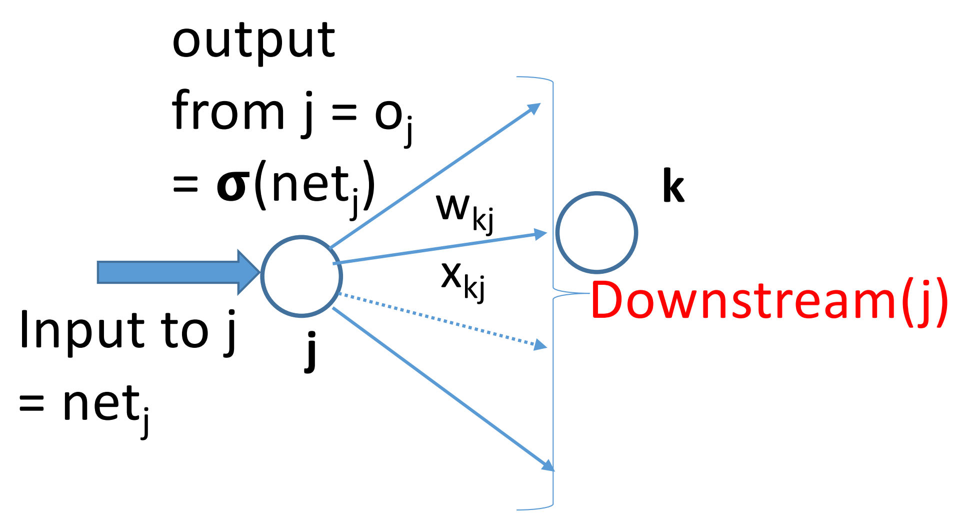
Case II: j is a hidden unit:

Note that net_j can influence the error only through downstream units:

$$\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j}$$



Backpropagation Algorithm

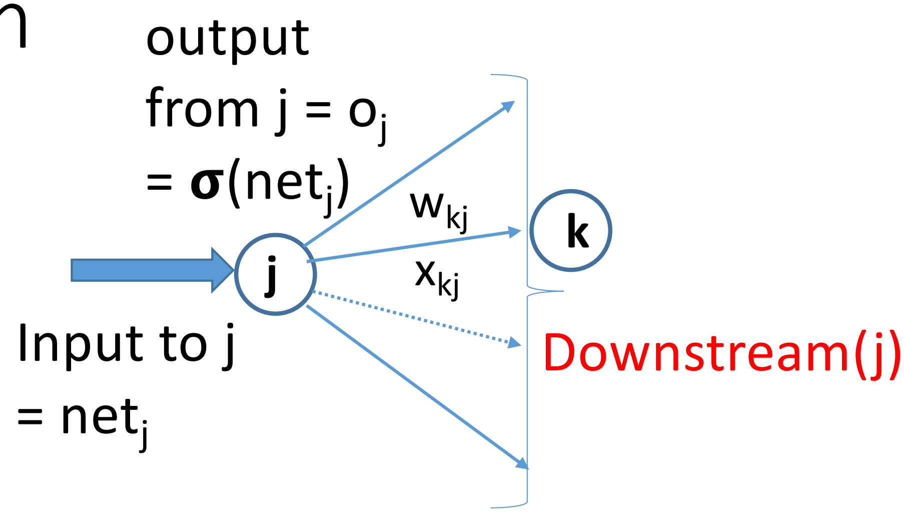
Case II: j is a hidden unit:

$$\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k w_{kj} o_j (1 - o_j)$$

Putting it all together:

$$\delta_j = o_j(1 - o_j) \sum_{k \in Downstream(j)} \delta_k w_{kj}$$



Plugging it in original equation:

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$
$$= \eta \delta_j x_{ji}$$

Backpropagation Summary

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

- Case I: j is output layer node:

$$\delta_j = (t_j - o_j) o_j (1 - o_j)$$

where x_{ji} is the weight of edge from i^{th} node to j^{th} node, where j is the output node.

- Case II: j is hidden layer node:

$$\delta_j = o_j (1 - o_j) \sum_{k \in Downstream(j)} \delta_k w_{kj}$$

Unit j sends its output to Downstream(j) units. δ_k is the delta for the unit k which is part of Downstream(j).

Backpropagation Algorithm

Initialize all weights to small random numbers

Until convergence, Do

For each training example, Do

1. Input it to network and compute network outputs — Forward pass
2. For each output unit k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

3. For each hidden unit h

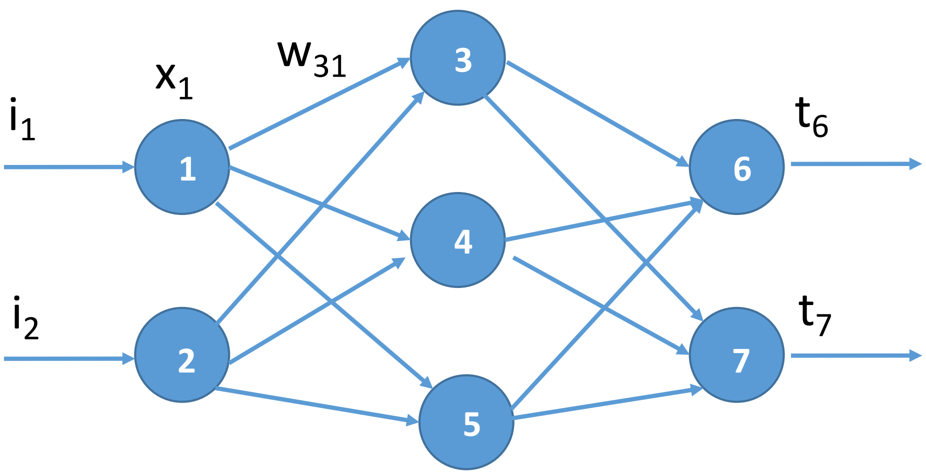
$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k} \delta_k$$

4. Update each network weight $w_{i,j}$ — Backward pass

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

where $\Delta w_{i,j} = \eta \delta_j x_{i,j}$

Example:



Assume random value for all weights on network:

Forward Pass:

Compute all outputs of the nodes.

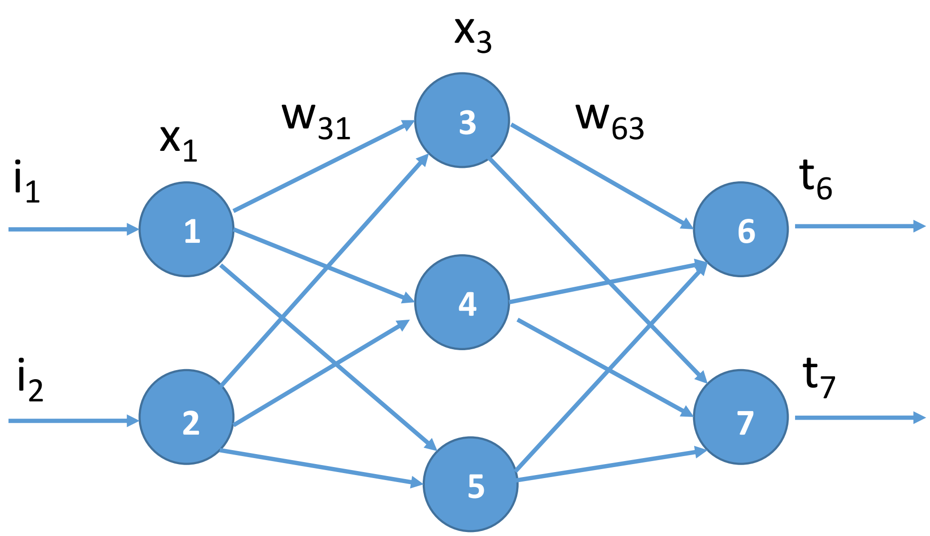
Find δ for output nodes

Backward Pass:

Use the value of forward deltas to compute backward deltas

Update weights.

Example:

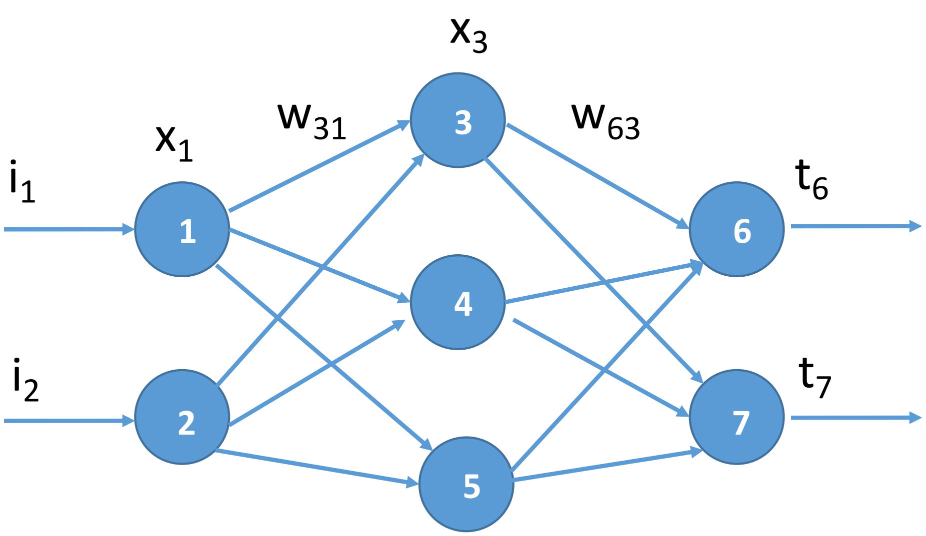


Forward Pass:

Node	Net	Output
1	i_1	$x_1 = i_1$ *
2	i_2	$x_2 = i_2$ *
3	$net_3 = w_{31}x_1 + w_{32}\, x_2$	$x_3 = \boldsymbol{\sigma}(net_3)$
4	$net_4 = w_{41}x_1 + w_{42}\, x_2$	$x_4 = \boldsymbol{\sigma}(net_4)$
5	$net_5 = w_{51}x_1 + w_{52}\, x_2$	$x_5 = \boldsymbol{\sigma}(net_5)$
6	$net_6 = w_{63}x_3 + w_{64}\, x_4 + w_{65}\, x_5$	$x6 = \boldsymbol{\sigma}(net_6)$
7	$net_7 = w_{73}x_3 + w_{74}\, x_4 + w_{75}\, x_5$	$x7 = \boldsymbol{\sigma}(net_7)$

* Most books assume $x_1 = i_1$ (sigmoid not applied to input)
and others assume $x_1 = \boldsymbol{\sigma}(i_1)$ (sigmoid is applied to input)

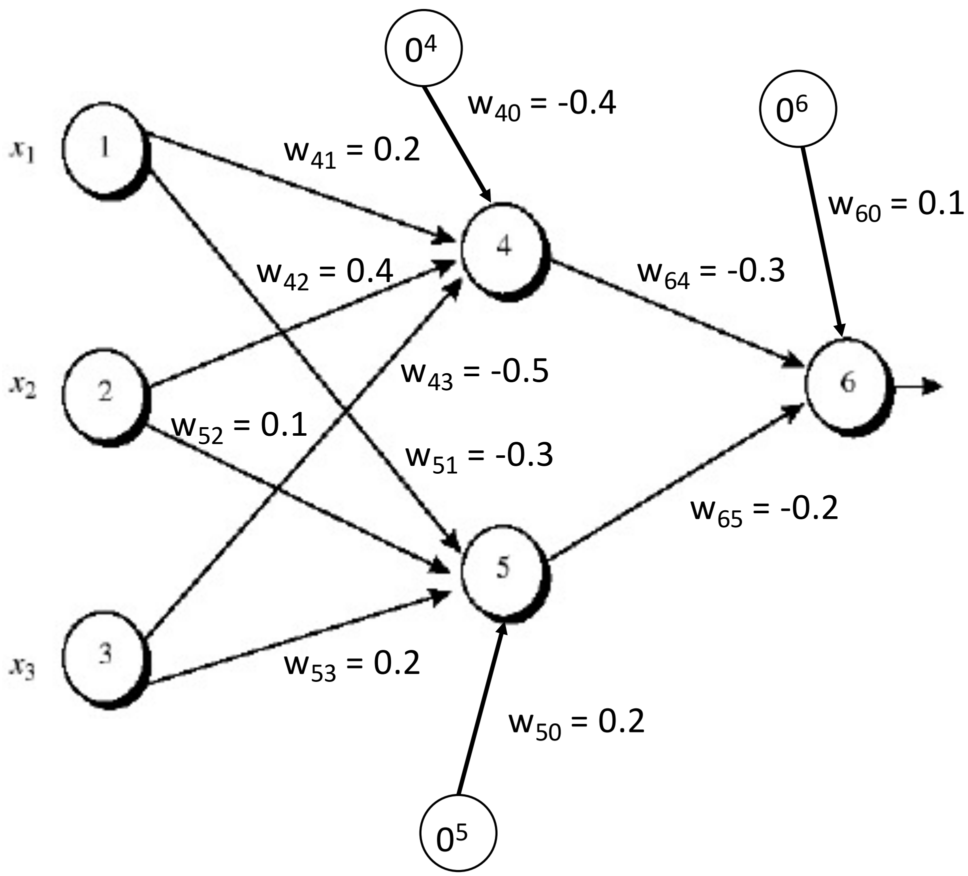
Example:



Backward Pass:

Node	Delta	Weight Update
7	$\delta_7 = x_7\,(1\text{-}x_7)\,(t_7 - x_7)$	
6	$\delta_6 = x_6\,(1\text{-}x_6)\,(t_6 - x_6)$	
5	$\delta_5 = x_5\,(1\text{-}x_5)\,[w_{65}\delta_6 + w_{75}\delta_7]$	$\Delta w_{65} = \boldsymbol{\eta}\,\delta_6\,x_5$ $\Delta w_{75} = \boldsymbol{\eta}\,\delta_7\,x_5$
4	$\delta_4 = x_4\,(1\text{-}x_4)\,[w_{64}\delta_6 + w_{74}\delta_7]$	$\Delta w_{64} = \boldsymbol{\eta}\,\delta_6\,x_4$ $\Delta w_{74} = \boldsymbol{\eta}\,\delta_7\,x_4$
3	$\delta_3 = x_3\,(1\text{-}x_3)\,[w_{63}\delta_6 + w_{73}\delta_7]$	$\Delta w_{63} = \boldsymbol{\eta}\,\delta_6\,x_3$ $\Delta w_{73} = \boldsymbol{\eta}\,\delta_7\,x_3$
2	$\delta_2 = x_2\,(1\text{-}x_2)\,[w_{52}\delta_5 + w_{42}\delta_4 + w_{32}\delta_3]$	$\Delta w_{32} = \boldsymbol{\eta}\,\delta_3\,x_2$ $\Delta w_{42} = \boldsymbol{\eta}\,\delta_4\,x_2$ $\Delta w_{52} = \boldsymbol{\eta}\,\delta_5\,x_2$
1	$\delta_1 = x_1\,(1\text{-}x_1)\,[w_{51}\delta_5 + w_{41}\delta_4 + w_{31}\delta_3]$	$\Delta w_{31} = \boldsymbol{\eta}\,\delta_3\,x_1$ $\Delta w_{41} = \boldsymbol{\eta}\,\delta_4\,x_1$ $\Delta w_{51} = \boldsymbol{\eta}\,\delta_5\,x_1$

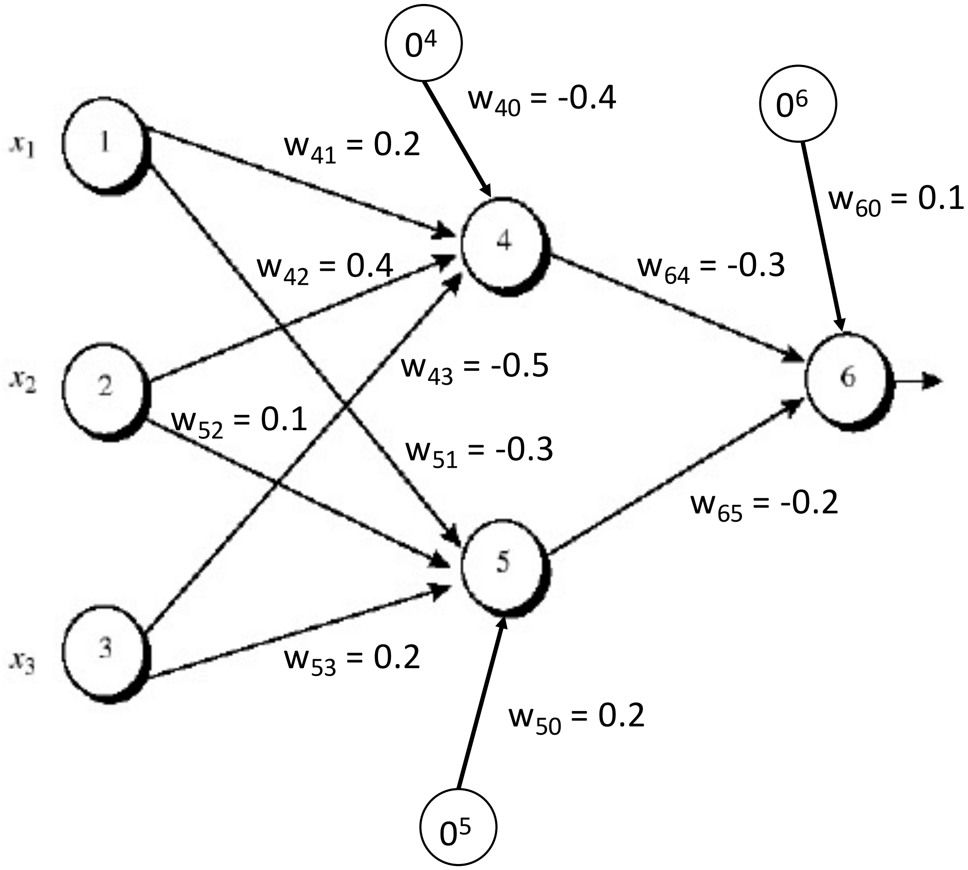
Worked Out Example



- The input to the ANN is (1, 0, 1) and target output is 1.

  $i_1 = 1$, $i_2 = 0$, $i_3 = 1$, and $t_6 = 1$
- Run one forward and one backward pass on the ANN

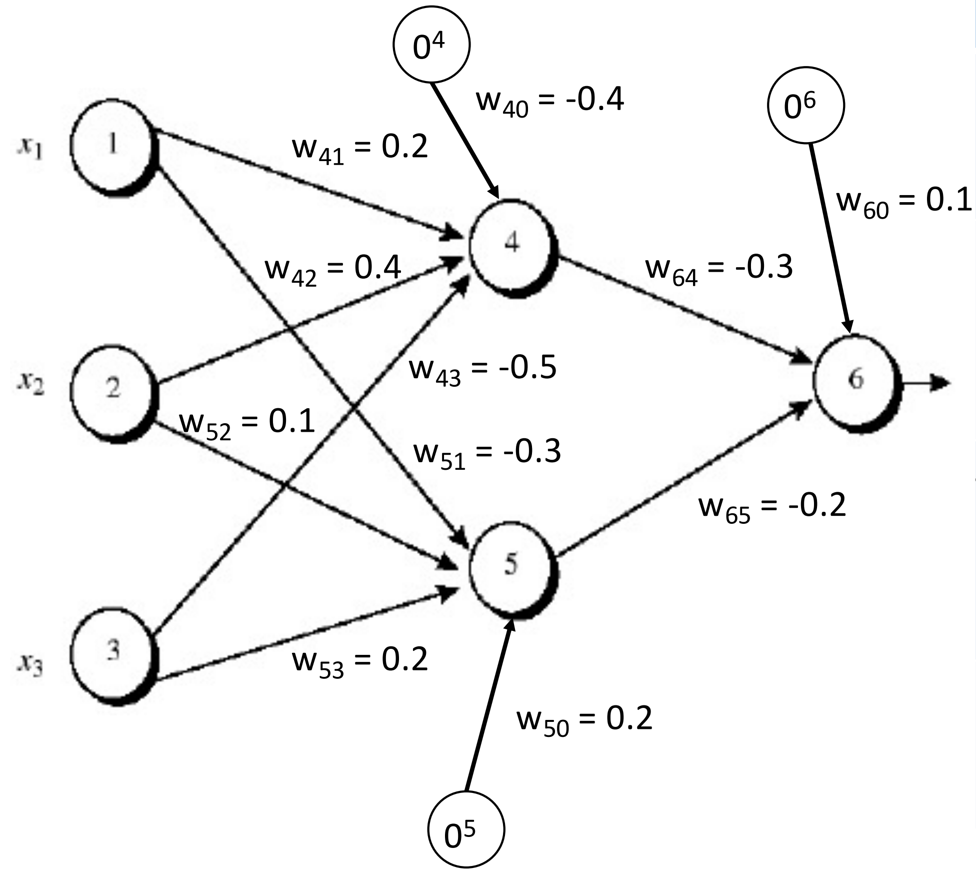
Worked Out Example



Forward Pass:

Node	Net	Output
1	i_1	$x_1 = 1$
2	i_2	$x_2 = 0$
3	i_3	$x_3 = 1$
4	$net_4 = w_{40}x_0 + w_{41}x_1 + w_{42}\,x_2 + w_{43}\,x_3$ $= -0.4 * 1 + 0.2 * 1 + 0.4 * 0 - 0.5 * 1$ $= -0.7$	$x_4 = \boldsymbol{\sigma}(net_4)$ $= 0.332$
5	$net_5 = w_{50}x_0 + w_{51}x_1 + w_{52}\,x_2 + w_{53}\,x_3$ $= 0.2 * 1 - 0.3 * 1 + 0.1 * 0 + 0.2 * 1$ $= 0.1$	$x_5 = \boldsymbol{\sigma}(net_5)$ $= 0.525$
6	$net_6 = w_{60}x_0 + w_{64}x_4 + w_{65}\,x_5$ $= 0.1 * 1 - 0.3 * 0.332 - 0.2 * 0.525$ $= -0.105$	$x6 = \boldsymbol{\sigma}(net_6)$ $= 0.474$

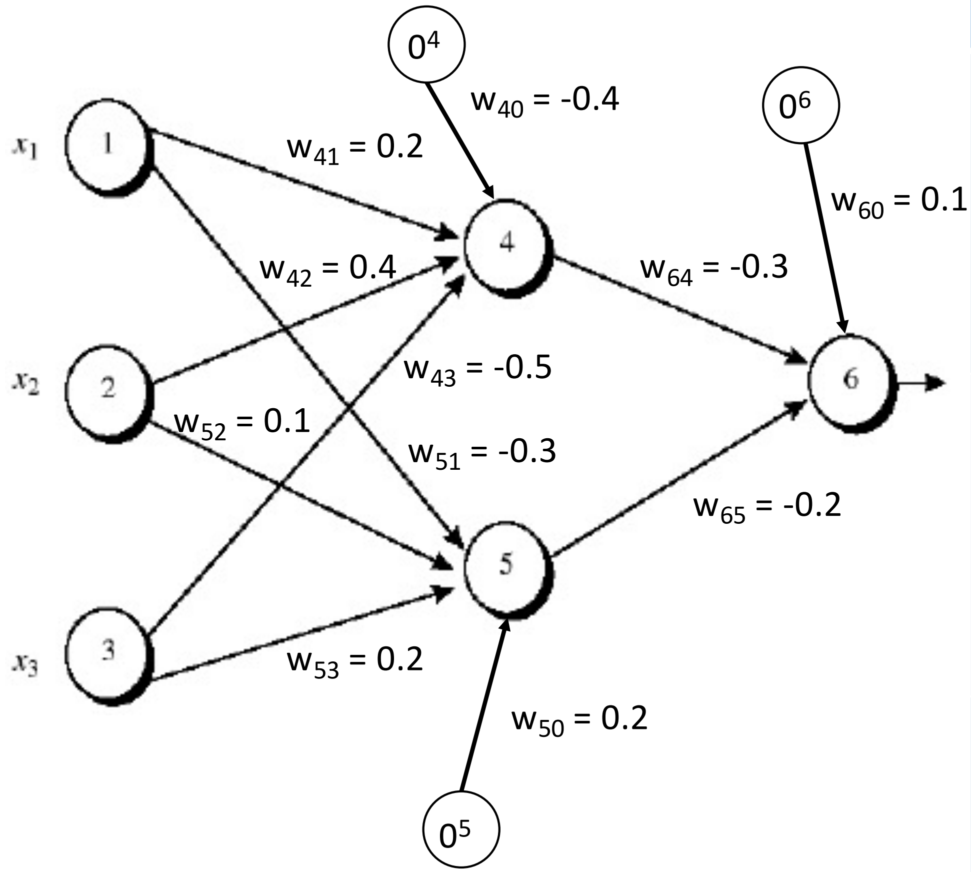
Worked Out Example



Backward Pass:

Node	Delta	Weight Update
6	$\delta_6 = x_6 (1-x_6) (t_6 - x_6)$ = 0.474 * (1- 0.474) * (1 – 0.474) = 0.131	
5	$\delta_5 = x_5 (1-x_5) (w_{65}\delta_6)$ = 0.525 * (1 – 0.525) * (-0.2 * 0.131) = -0.006	$\Delta w_{65} = \boldsymbol{\eta}\ \delta_6 x_5$ = 0.9 * 0.131 * 0.525 = 0.062
4	$\delta_4 = x_4 (1-x_4) (w_{64}\delta_6)$ = 0.332 * (1 – 0.332) * (-0.2 * 0.131) = -0.006	$\Delta w_{64} = \boldsymbol{\eta}\ \delta_6 x_4$ = 0.9 * 0.131 * 0.332 = 0.039
0^6	δ_{0^6} = 0 [because $x_{0^6} = 1$]	$\Delta w_{60}{}^6 = \boldsymbol{\eta}\ \delta_6 x_0{}^6$ = 0.9 * 0.131 * 1 = 0.118

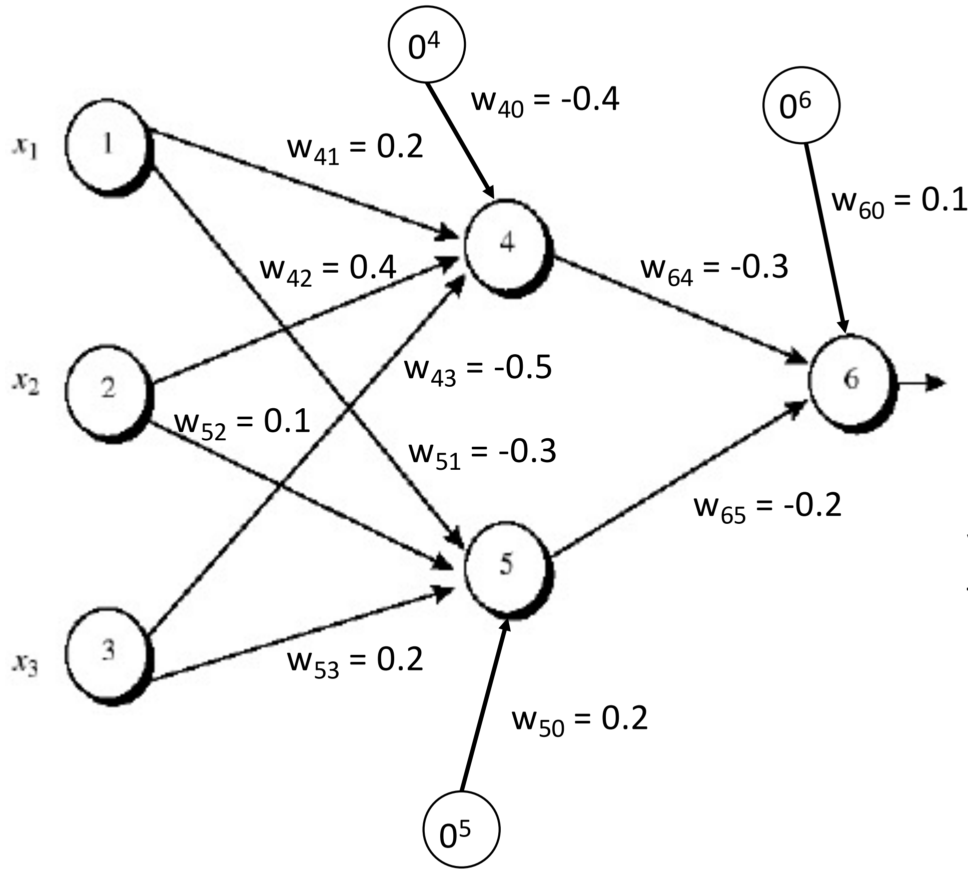
Worked Out Example



Backward Pass:

Node	Delta	Weight Update
3	No need to compute	$\Delta w_{43} = \boldsymbol{\eta}\ \delta_4 x_3$ = 0.9 * -0.006 * 1 = - 0.005 $\Delta w_{53} = \boldsymbol{\eta}\ \delta_5 x_3$ = 0.9 * -0.006 * 1 = - 0.005
2	No need to compute	$\Delta w_{42} = \boldsymbol{\eta}\ \delta_4 x_2$ = 0.9 * -0.006 * 0 = 0 $\Delta w_{52} = \boldsymbol{\eta}\ \delta_5 x_2$ = 0.9 * -0.006 * 0 = 0
1	No need to compute	$\Delta w_{41} = \boldsymbol{\eta}\ \delta_4 x_1$ = 0.9 * -0.006 * 1 = -0.005 $\Delta w_{51} = \boldsymbol{\eta}\ \delta_5 x_1$ = 0.9 * -0.006 * 1 = -0.005

Worked Out Example



Backward Pass:

Node	Delta	Weight Update
0^5	No need to compute	$\Delta w_{50}^{5} = \boldsymbol{\eta}\ \delta_5\, x_0^{5}$ $= 0.9 * -0.006 * 1$ $= -0.005$
0^4	No need to compute	$\Delta w_{40}^{4} = \boldsymbol{\eta}\ \delta_4\, x_0^{4}$ $= 0.9 * -0.006 * 1$ $= -0.005$

Finally, you update each of the weights as:

$w_{ij} = w_{ij} + \Delta w_{ij}$

for all applicable pairs (i, j)

More on Backpropagation

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often works well
    (can run multiple times)
- Often include weight *momentum* α

$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n-1)$$

The update at step n depends on step n-1

- Minimizes error over *training* examples
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations $\rightarrow$ slow!
- Using network after training is very fast

Expressiveness of Neural Nets

Boolean functions:

- Every Boolean function can be represented by network with single hidden layer
- But might require exponential (in number of inputs) hidden units

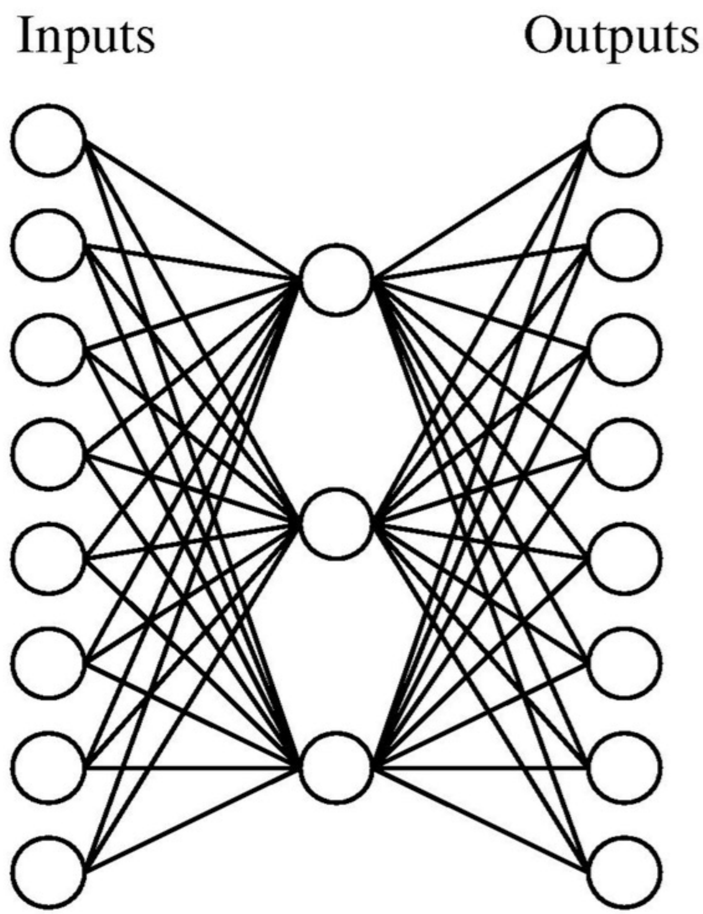
Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers

Learning Hidden Layer Representations

What is the role of hidden layers?

- To find a representation of the input and present it to the output layers.
- The representation can be compact.



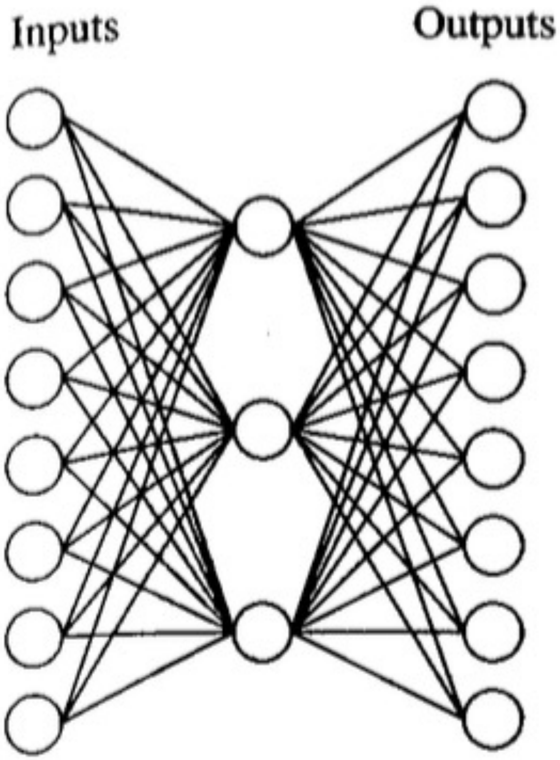
A target function:

Can the NN learn this function?
- Well, it's just the identity function
- Yes, with one hidden layer.
- The hidden layer will try to find a compact representation of the input
- Let's run the backprop algorithm on it and see what weights we get.

Input		Output
10000000	$\rightarrow$	10000000
01000000	$\rightarrow$	01000000
00100000	$\rightarrow$	00100000
00010000	$\rightarrow$	00010000
00001000	$\rightarrow$	00001000
00000100	$\rightarrow$	00000100
00000010	$\rightarrow$	00000010
00000001	$\rightarrow$	00000001

Can this be learned?

Learned hidden layer representation:

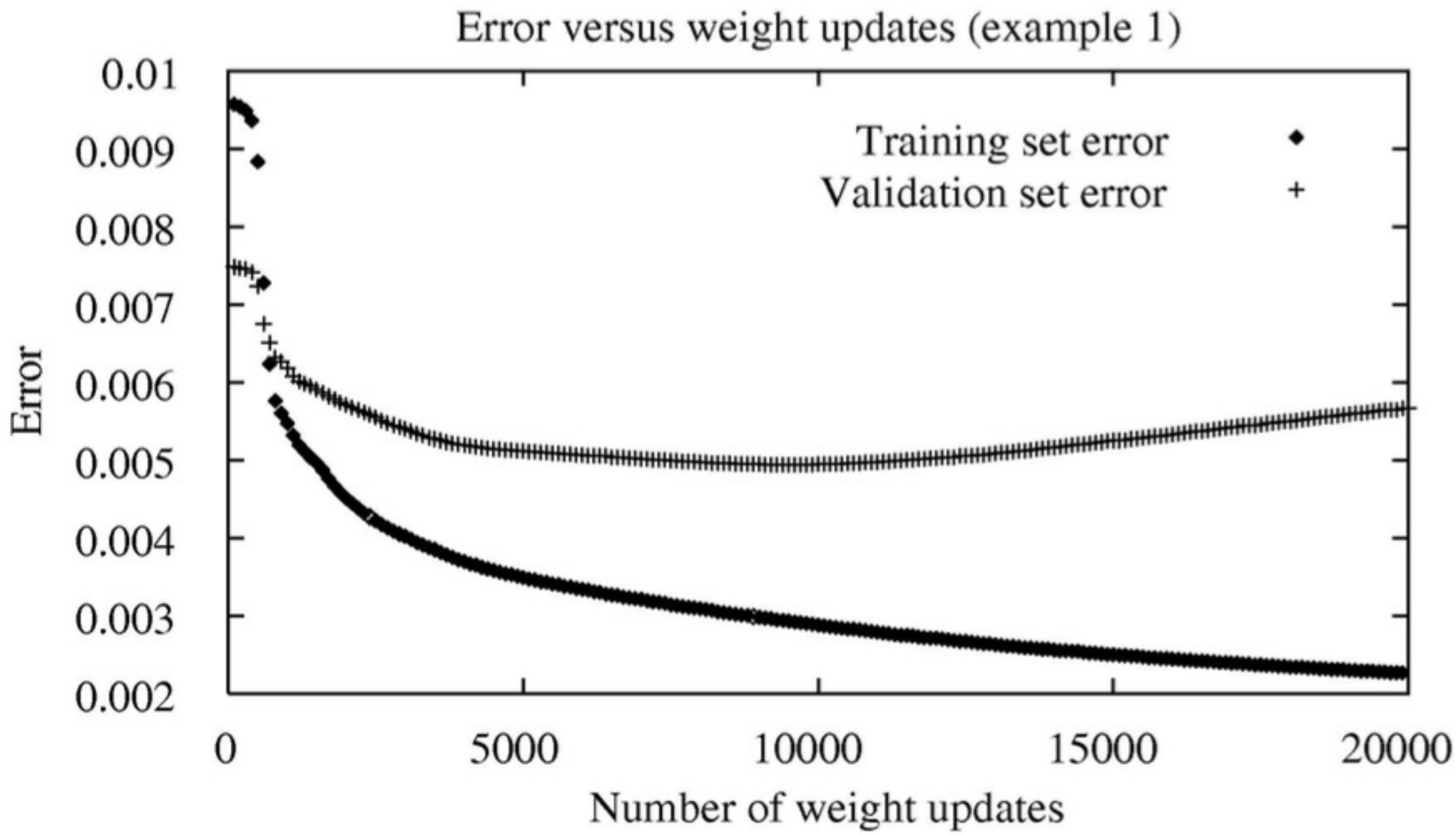


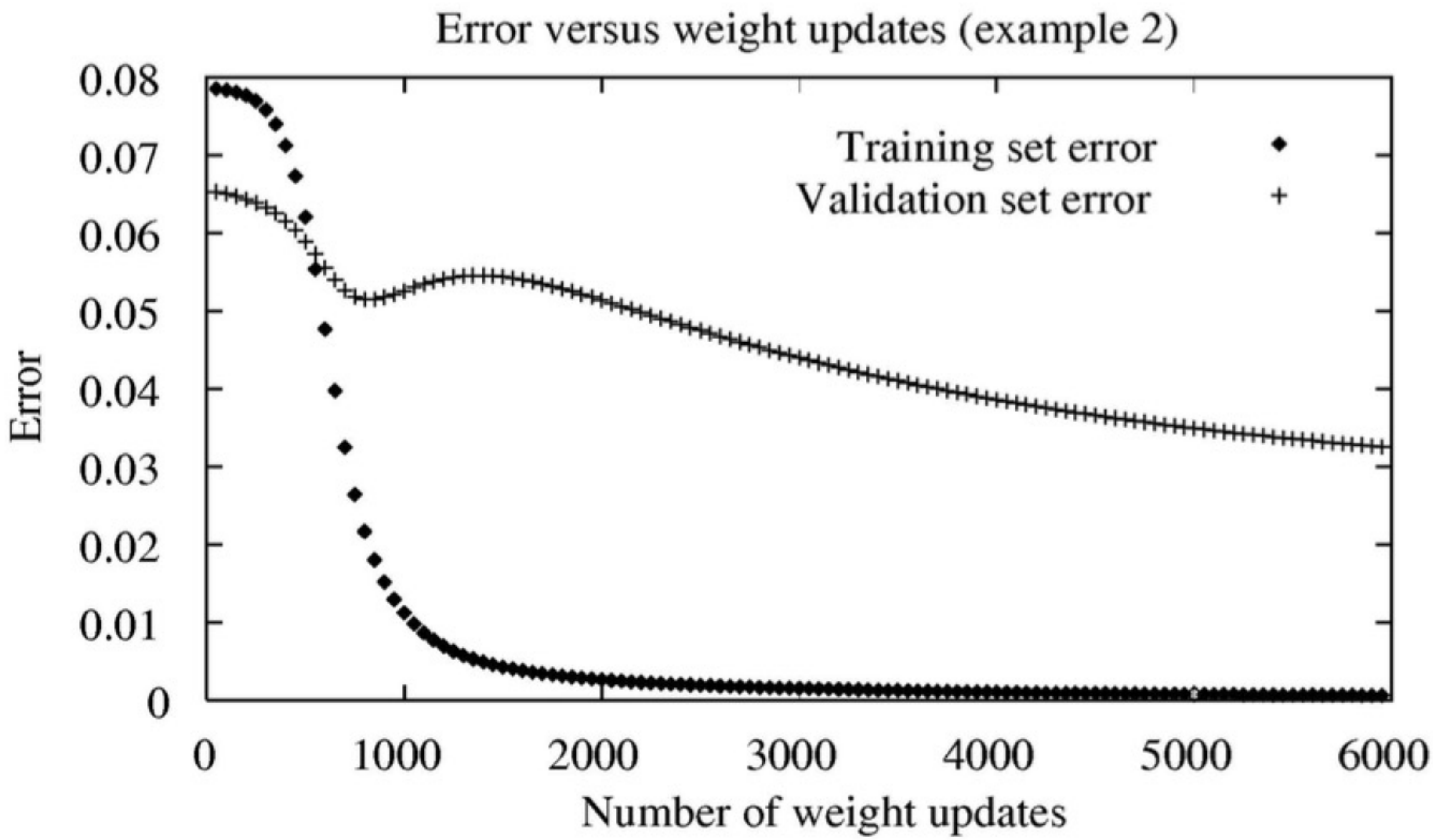
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

You can represent numbers 1 to 8 using 3 Boolean bits as (001) to (111)

Note: The weights do not necessarily correspond to human interpretation. The NN has its own encoding mechanism

Overfitting in Neural Nets





Error versus weight updates (example 2)
Training set error
Validation set error
Error
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
0
1000
2000
3000
4000
5000
6000
Number of weight updates

Overfitting Avoidance

Penalize large weights:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

Train on target slopes as well as values:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} \left[(t_{kd} - o_{kd})^2 + \mu \sum_{j \in inputs} \left(\frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$

Weight sharing

Early stopping