

**CS 4700:  
Foundations of Artificial Intelligence**

**Prof. Bart Selman**  
[selman@cs.cornell.edu](mailto:selman@cs.cornell.edu)

**Machine Learning:  
Neural Networks  
R&N 18.7  
Backpropagation**

# Backpropagation

In order to learn multi-layer neural nets,  
we need another learning algorithm.  
(invented ca. 1984)

A multi-layer net has one or more hidden layers.

We will consider the **backpropagation**  
algorithm for training such networks.

See also R&N. Here we will present a more  
detailed example.

Based on Nilsson (Stanford)

Note that in the perceptron case, we looked at the output value, compared it to the desired value and changed the weights accordingly.

We want to do something similar in the multi-layer case but one difficulty is to determine the error in the outputs of the hidden units.

Luckily, we can approximate those errors by “backpropagating” the final output error.

To do so, we need an activation function that is differentiable.

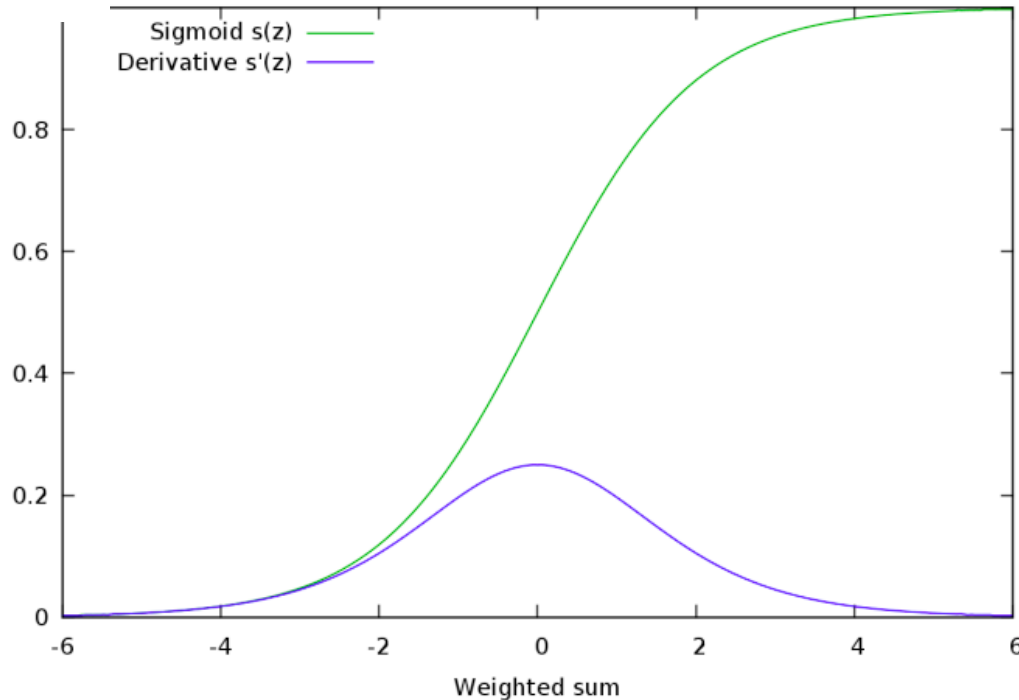
We use the **sigmoid** function to compute our output values.

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

The derivative of  $f(x)$  is:

$$f'(x) = f(x) \times (1 - f(x))$$

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



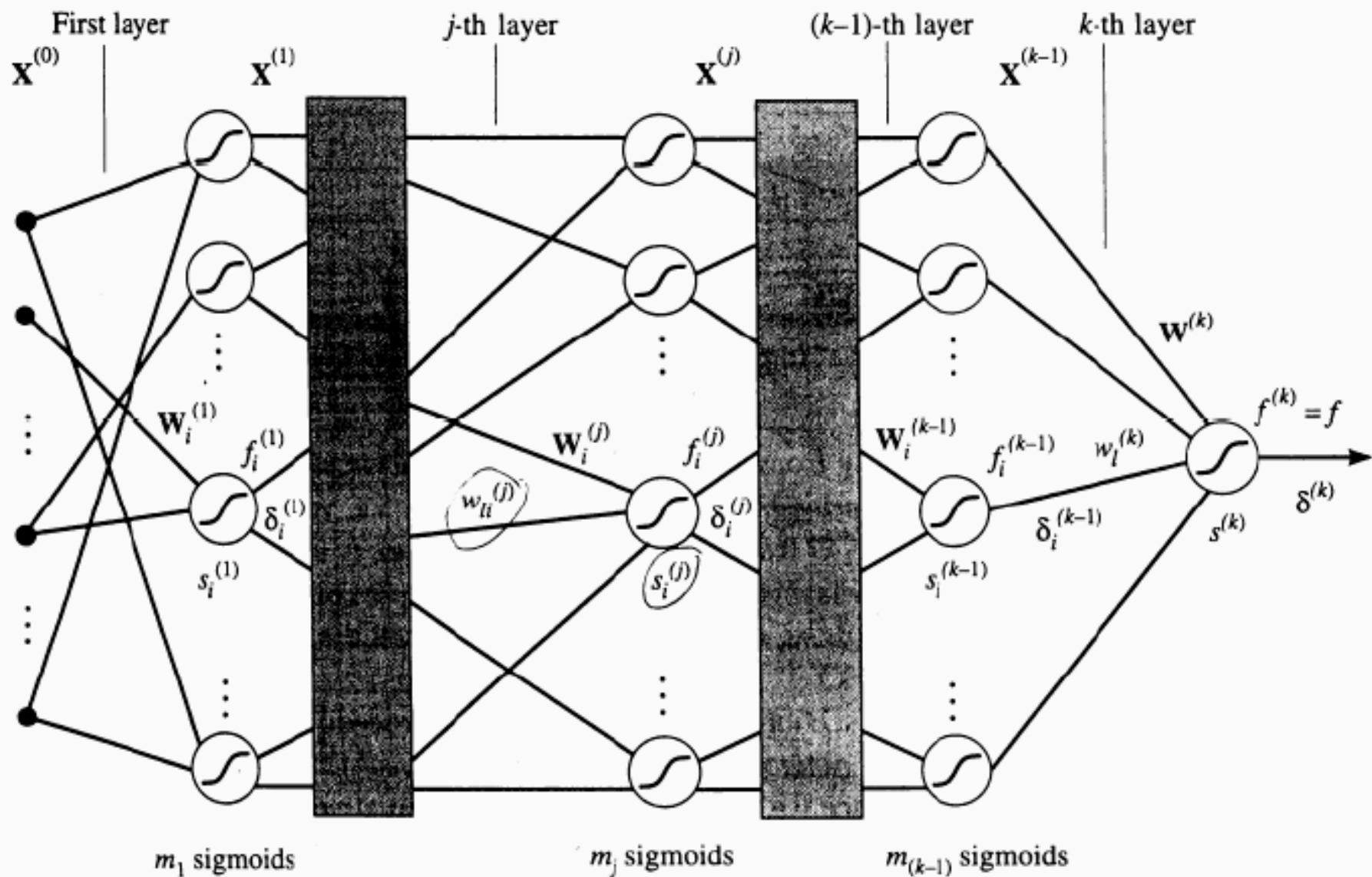
Note: largest derivative at  $x = 0$

That's where neuron is most sensitive to weight changes (effect of changes is well “controlled”).

Output value of neuron is simply the weighted sum of its input “pushed” through the sigmoid.

$$f(s) = \frac{1}{1+e^{-s}}, \text{ where}$$
$$s = \sum_{k=1}^{k=n+1} w_k \times x_k$$

Again, we assume the threshold is replaced by an extra input fixed at 1. Inputs: 0 or 1.



**Figure 3.5**

A  $k$ -layer Network of Sigmoid Units

**Ok... *details* backpropagation  
NOT on exam.**

**But do work through gradient  
descent example on homework.**

Setup and notation:

$k$ -layer network.

Input vector:

$$\vec{X}^{(0)} = \langle x_1^{(0)}, x_2^{(0)}, \dots, x_{m_0}^{(0)} \rangle$$

The first layer has  $m_1$  units and its output is:

$$\vec{X}^{(1)} = \langle f_1^{(1)}, f_2^{(1)}, \dots, f_{m_1}^{(1)} \rangle$$

The weights for the  $i$ th unit in the first layer are  
given by:

$$\vec{W}_i^{(1)} = \langle w_{1,i}^{(1)}, w_{2,i}^{(1)}, \dots, w_{m_0,i}^{(1)} \rangle$$



These outputs  $\vec{X}^{(1)}$  are fed into the second layer of  $m_2$  units.

The number of units in the  $j$ -th layer is  $m_j$ .

In general, the weight vector of unit  $i$  in the  $j$ -th layer is  $\vec{W}_i^{(j)}$  with components  $w_{l,i}^j$  for  $l = 1, m_{(j-1)} + 1$ .

Note that the previous layer had  $m_{(j-1)}$  units (and thus outputs) all connected to each unit in the  $j$ -th layer. We use one additional weight and fixed input to model the threshold. (Not given on previous slide.)

The weighted sum of the inputs to  $i$ -th sigmoid unit in the  $j$ -th layer is denoted by  $s_i^{(j)}$

$$\text{The output is } f_i^{(j)} = \frac{1}{1+e^{-s_i^{(j)}}}$$

We have:

$$s_i^{(j)} = \vec{X}^{(j-1)} \cdot \vec{W}^{(j)}$$

We used the vector dot product, i.e.,

$$s_i^{(j)} = \sum_{l=1}^{l=m_{(j-1)}+1} x_l^{(j-1)} \times w_{l,j}^{(j)}$$

$x_l^{(j-1)}$  is the output of unit  $l$  in the

previous layer. It's connected with weight  $w_{l,i}^{(j)}$  to the  $i$ -th unit in the  $j$ -th layer.

Note: again, concerning the “+1” here, we assume that an extra “1” is added to the input vector and an extra weight, to model the threshold.

Finally, the  $k$ -th layer is the output layer.

It has a single unit with input  $s^{(k)}$  (weighted sum) and output value  $f^{(k)} = f$ .

The objective of the backpropagation algorithm is to minimize the output error on each example.

That is, we want to minimize:

$$(L - f)^2$$

Where,  $L$  the label (0 or 1) of the input example under consideration and  $f$  is the output of the network given that example.

Note: we will update the weights after each example

An alternative approach considers the combined error over the total training set and update after seeing all examples. In the limit the approaches are the same.

A key observation is that the error  $(L - f)^2$  is only a function of the weights.

Note: the number of units etc. is fixed.

Also, the inputs are fixed, since we are considering a particular example.

The idea is now to do a **gradient** descent in the weight space to minimize the error.

The derivation (basically calculus) is somewhat involved but not difficult.

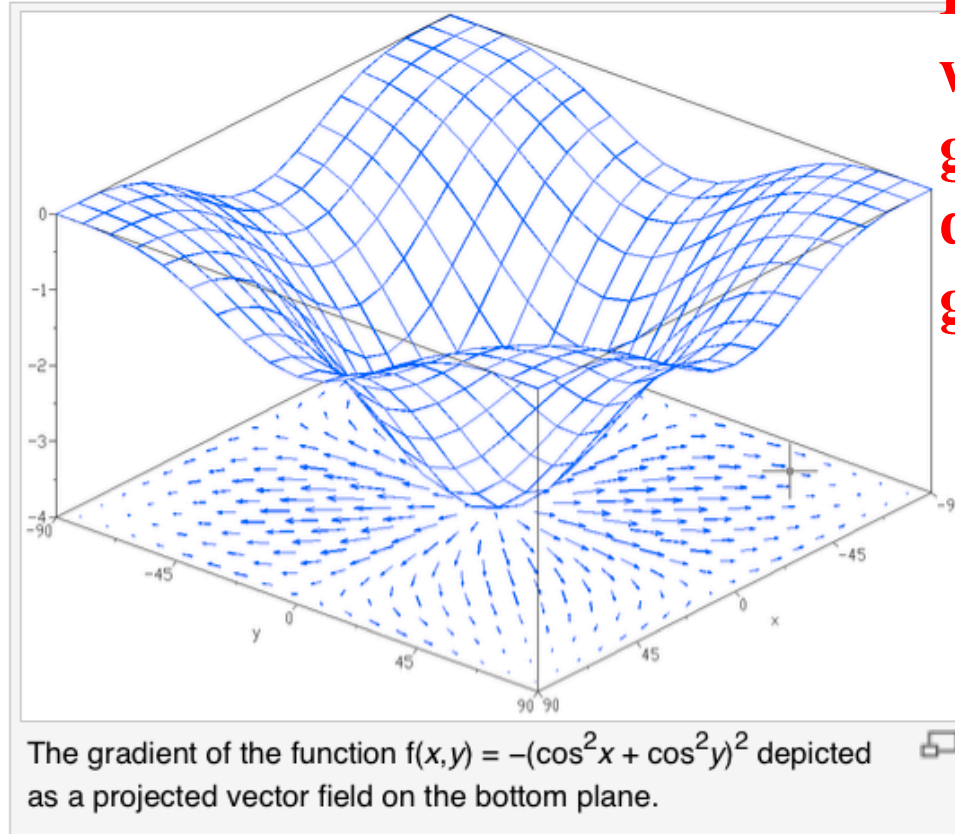
**It's “just” multivariate (or multivariable) calculus.**

# Optimization and gradient descent.

Consider  $x$  and  $y$  our two weights and  $f(x,y)$  the error signal.

Multi-layer network: composition of sigmoids; use chain rule.

**Note:**  
For decent we want to go in opposite direction of gradient.



$$\nabla f = \frac{\partial f}{\partial x_1} \mathbf{e}_1 + \dots + \frac{\partial f}{\partial x_n} \mathbf{e}_n$$

Gradient descent is based on the observation that if the multivariable function  $F(\mathbf{x})$  is **defined** and **differentiable** in a neighborhood of a point  $\mathbf{a}$ , then  $F(\mathbf{x})$  decreases *fastest* if one goes from  $\mathbf{a}$  in the direction of the negative gradient of  $F$  at  $\mathbf{a}$ ,  $-\nabla F(\mathbf{a})$ . It follows that, if

$$\mathbf{b} = \mathbf{a} - \gamma \nabla F(\mathbf{a})$$

for  $\gamma$  small enough, then  $F(\mathbf{a}) \geq F(\mathbf{b})$ . With this observation in mind, one starts with a guess  $\mathbf{x}_0$  for a local minimum of  $F$ , and considers the sequence  $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \dots$  such that

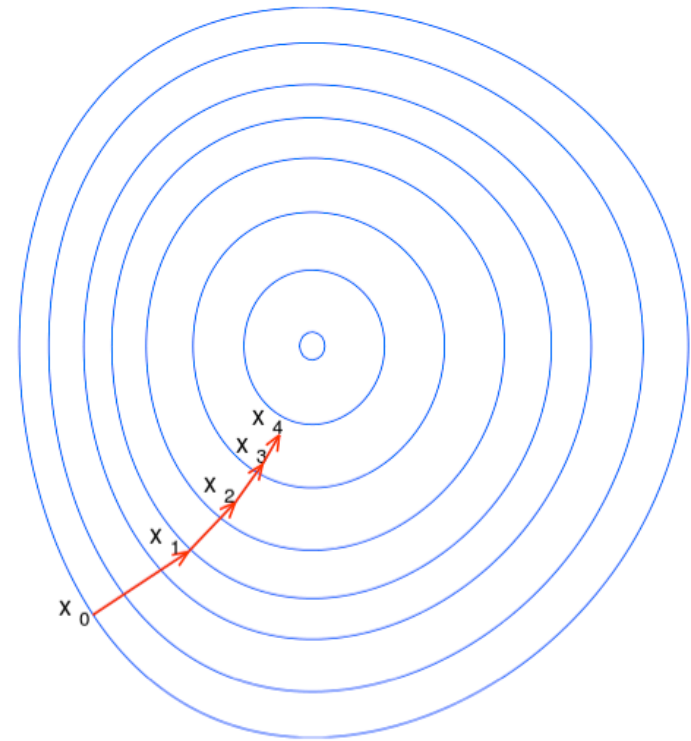
$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla F(\mathbf{x}_n), \quad n \geq 0.$$

We have

$$F(\mathbf{x}_0) \geq F(\mathbf{x}_1) \geq F(\mathbf{x}_2) \geq \dots,$$

so hopefully the sequence  $(\mathbf{x}_n)$  converges to the desired local minimum.

Reminder:  
Gradient descent optimization.  
(Wikipedia)



Here we just give the results of the calculation.

First, the weight adjustment for the single unit in the output layer is given by:

$$\vec{W}^{(k)} \leftarrow \vec{W}^{(k)} + \alpha \times (d - f) \times f \times (1 - f) \times \vec{X}^{(k-1)}$$

Note:  $f$  is the output of the unit;  $d$  is desired output.

This rule is analogous to the perceptron rule, except that we are now using the sigmoid.

The  $(d - f)$  is the error signal.

$\alpha$  is the learning rate (a constant chosen by the user).

$f \times (1 - f)$  comes from the derivative of the sigmoid.

$\vec{X}^{(k-1)}$  is the input to the unit under consideration.



So, again we're adding (subtracting) the input vector, depending on  $(d - f)$ .

Please check for yourself that the correction is in the right direction!

We're basically going to do something similar for the weights in the hidden layer. The only problem is that we don't have a direct measure of the error in the outputs on the hidden units.

However, we can estimate those errors by considering the contribution (an estimate) of each unit to the final output value (layer  $k$ ).

Starting with the final layer and moving backwards,  
we compute for the  $i$ -th unit in the  $j$ -th layer:

$$\delta_i^{(j)} = f_i^j (1 - f_i^j) \sum_{l=1}^{m_{j+1}} \delta_l^{(j+1)} w_{i,l}^{(j+1)}$$

Note that  $f_i^{(j)}$  is the output value of the unit.

So, the  $\delta$  for a unit in the  $i$ -th layer is

computed by considering the delta in the  $i + 1$ -th layer.

The base case is the output layer  $k$ :

$$\delta^{(k)} = (d - f) \times f \times (1 - f)$$

I.e., the real output error times the gradient.

This values is propagated backwards to get error measures  
for the internal units (times the gradient again).

Finally, using these  $\delta$ 's, we can compute the weight updates for the hidden units:

$$\vec{W}_i^{(j)} \leftarrow \vec{W}_i^{(j)} + \alpha \times \delta_i^j \times \vec{X}^{(j-1)}$$

Verify that this rule is consistent with the update rule for the final node. (Check  $j = k$ ; drop  $i$ , since only one unit in layer.)

**So, first we “backpropagate” the error signal, and then we update the weights, layer by layer.**

**That’s why it’s called “backprop learning.”  
Now changing speech recognition, image recognition etc.!**

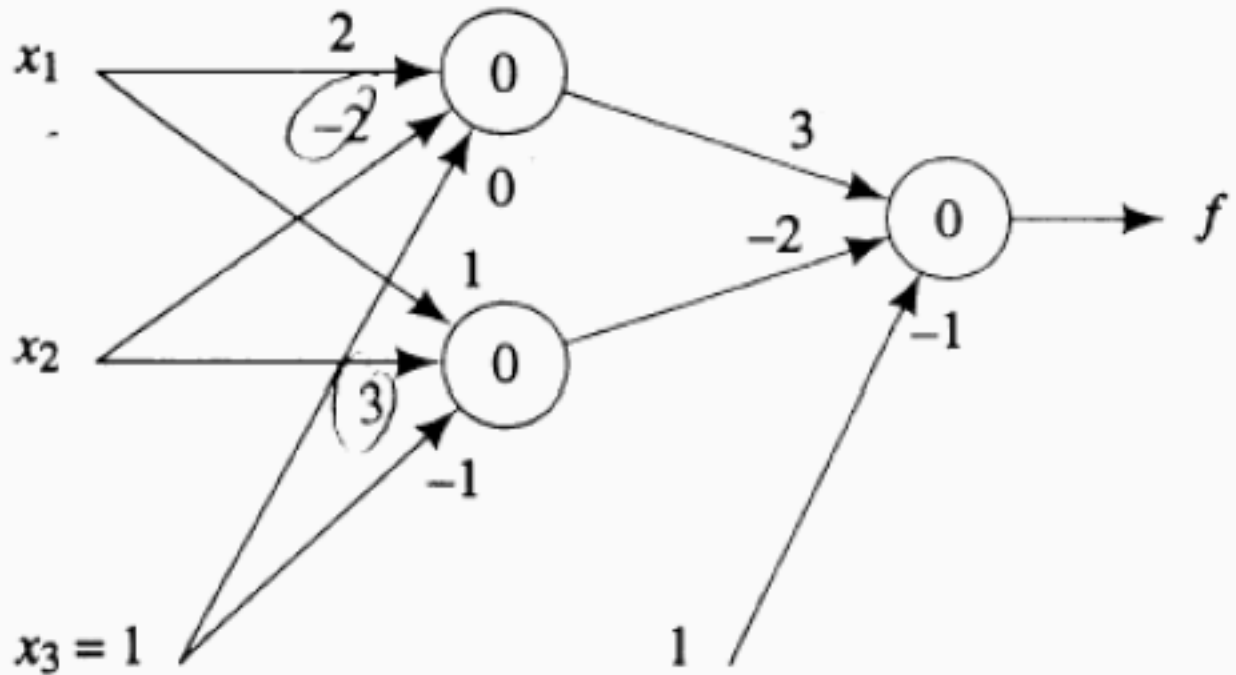
Although, the rules are somewhat intuitive, only by doing the full derivation, can one explain all the terms.

Let us consider an example of the procedure in action.

## Trace example!

**Figure 3.6**

A Network to Be Trained by Backprop



We have two layers ( $k = 2$ ).

Two units in first layer, with a total  
of three inputs  $(x_1^{(0)}, x_2^{(0)}, x_3^{(0)})$ .  
 $x_3^{(0)}$  will be fixed to 1.

The two units in the first layer are connected  
to a node in the final layer. This node  
has three inputs:  $(x_1^{(1)}, x_2^{(1)}, x_3^{(1)})$ .

$x_1^{(1)}$  is the output of the 1st unit in the 1st layer.

$x_2^{(1)}$  is the output of the 2nd unit in the 2nd layer.

$x_3^{(1)}$  is fixed at 1.

Note the given weights in the figure.

We want to train the network to capture the following patterns:

ex 1.)  $x_1^{(0)} = 1, x_2^{(0)} = 0, x_3^{(0)} = 1, d = 0$

ex 2.)  $x_1^{(0)} = 0, x_2^{(0)} = 0, x_3^{(0)} = 1, d = 1$

ex 3.)  $x_1^{(0)} = 0, x_2^{(0)} = 1, x_3^{(0)} = 1, d = 0$

ex 4.)  $x_1^{(0)} = 1, x_2^{(0)} = 1, x_3^{(0)} = 1, d = 1$

Again,  $d$  is the desired output;  $x_3^{(0)}$  is the fixed unit.

Let's consider the update after the first pattern.

Ex 1.) gives input vector  $\langle 1, 0, 1 \rangle$ , which leads via the sigmoid to the following output values:

$$f_1^{(1)} = \frac{1}{1+e^{-2}} = 0.881$$

$$f_2^{(1)} = \frac{1}{1+e^{-0}} = 0.5$$

$$f = \frac{1}{1+e^{-(3 \times 0.881 + (-2) \times 0.5 - 1)}} = 0.665$$



We now compute the values for the  $\delta$ 's.

First the base case:

$$\delta^{(2)} = (0 - 0.665) \times 0.665 \times (1 - 0.665) = -0.148$$

Backpropagating through the weights gives:

$$\delta_1^{(1)} = 0.881 \times (1 - 0.881) \times (-0.148 \times 3) = -0.047$$

$$\delta_2^{(1)} = 0.5 \times (1 - 0.5) \times (-0.148 \times -2) = 0.074$$

Double-check at least one of these!

After computing the  $\delta$ 's, we can now update the weights.  
(Use learning rate  $\alpha = 1$ .) We get for the new weights:

$$\vec{W}_1^{(1)} = \langle 1.953, -2.0, -0.047 \rangle$$

$$\vec{W}_2^{(1)} = \langle 1.074, 3.0, -0.926 \rangle$$

$$\vec{W}^{(2)} = \langle 2.870, -2.074, -1.148 \rangle$$

Aside: the original weights were:

$$\vec{W}_1^{(1)} = \langle 2.0, -2.0, 0.0 \rangle$$

$$\vec{W}_2^{(1)} = \langle 1.0, 3.0, -1.0 \rangle$$

$$\vec{W}^{(2)} = \langle 3.0, -2.0, -1.0 \rangle$$

Let's do an example calculation of the first weight vector.

$$\vec{W}_1^{(1)} = \langle 1.953, -2.0, -0.047 \rangle$$

$$\begin{aligned} w_{1,1}^{(1)} &= w_{1,1}^{(1)} + (1 \times \delta_1^1 \times x_1^{(0)}) \\ &= 2 + (1 \times (-0.047) \times 1) = 1.953 \end{aligned}$$

$$\begin{aligned} w_{1,2}^{(1)} &= w_{1,2}^{(1)} + (1 \times \delta_1^1 \times x_2^{(0)}) \\ &= -2 + (1 \times (-0.047) \times 0) = -2.0 \end{aligned}$$

$$\begin{aligned} w_{1,3}^{(1)} &= w_{1,3}^{(1)} + (1 \times \delta_1^1 \times x_3^{(0)}) \\ &= 0 + (1 \times (-0.047) \times 1) = -0.047 \end{aligned}$$

Let's do the calculation of the third weight vector.

$$\begin{aligned}\vec{W}(2) &= \langle 2.870, -2.074, -1.148 \rangle \\ w_1^{(2)} &= w_1^{(2)} + (1 \times \delta^2 \times x_1^{(1)}) \\ &= 3 + (1 \times (-0.148) \times 0.881) \\ &= 2.870\end{aligned}$$

$$\begin{aligned}w_2^{(2)} &= w_2^{(2)} + (1 \times \delta^2 \times x_2^{(1)}) \\ &= -2 + (1 \times (-0.148) \times 0.5) \\ &= -2.074\end{aligned}$$

$$\begin{aligned}w_3^{(2)} &= w_3^{(2)} + (1 \times \delta^2 \times x_3^{(1)}) \\ &= -1 + (1 \times (-0.148) \times 1) \\ &= -1.148\end{aligned}$$

# Summary ☺

## **A tour of AI:**

### **I) AI**

- motivation**
- intelligent agents**

### **II) Problem-Solving**

- search techniques**
- adversarial search**
- reinforcement learning**

# Summary, cont. 😊

## III) Knowledge Representation and Reasoning

- logical agents
- Boolean satisfiability (SAT) solvers
- first-order logic and inference

## V) Learning

- from examples
- decision tree learning (info gain)
- neural networks

The field has grown (and continues to grow) exponentially but you have now seen a good part!

**Have a great winter break!!!!**