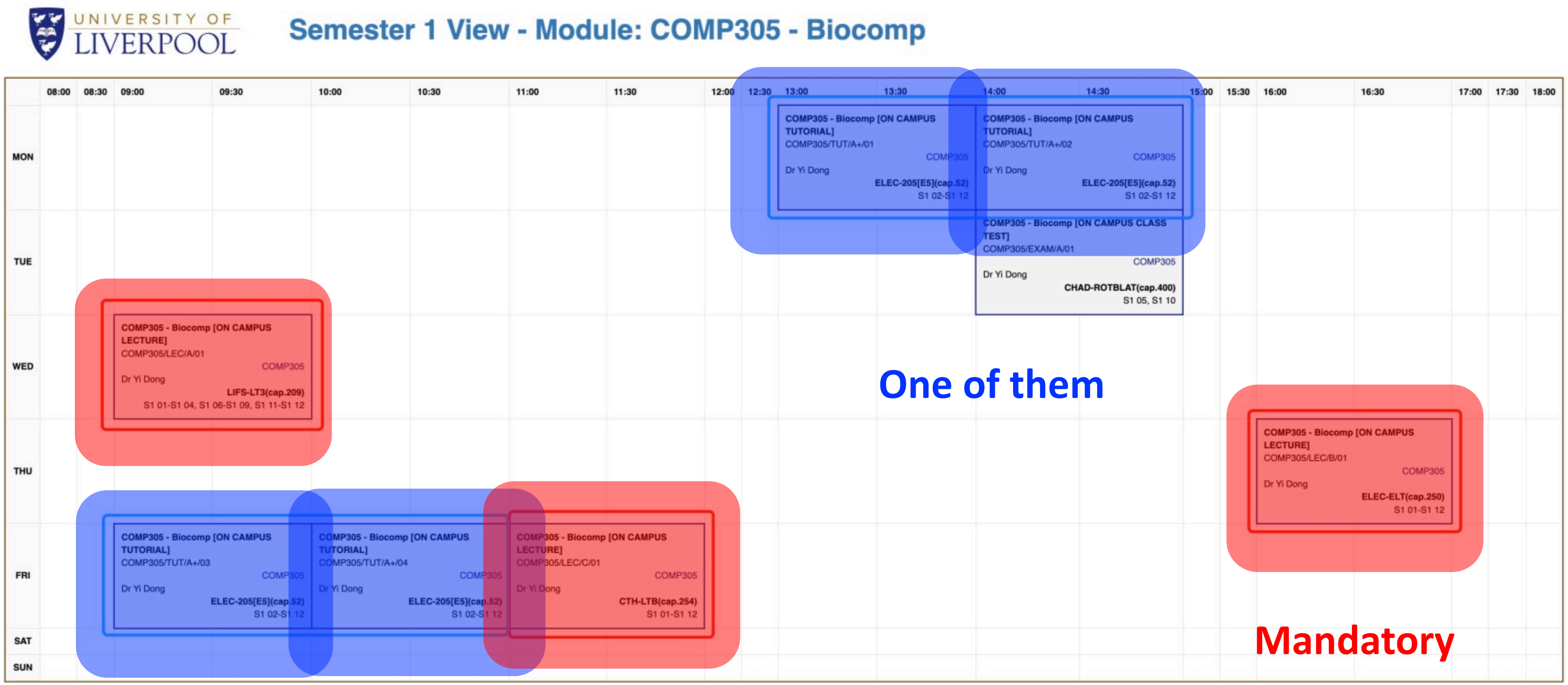
Comp305

Biocomputation

Lecturer: Yi Dong

Comp305 Module Timetable



There will be **26-30** lectures, thee per week. The lecture slides will appear on Canvas. Please use Canvas to access the lecture information. There will be **9** tutorials, one per week.

Lecture/Tutorial Rules

Questions are welcome as soon as they arise, because

1. Questions give feedback to the lecturer;

2. Questions help your understanding;

3. Your questions help your classmates, who might experience difficulties with formulating the same problems/doubts in the form of a question.

Comp305 Part I.

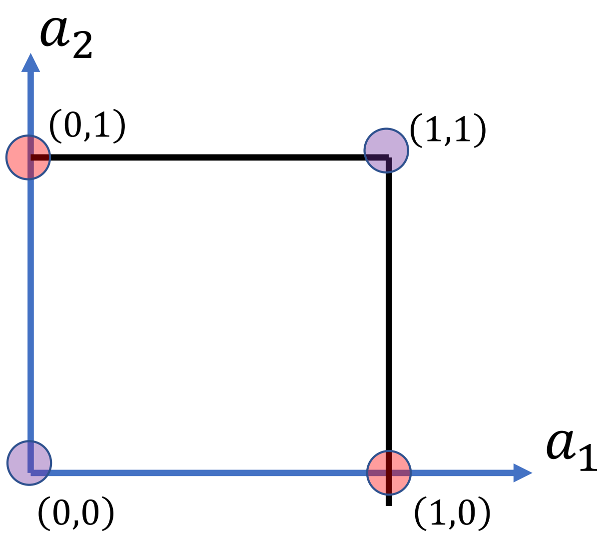
Artificial Neural Networks

Topic 5.

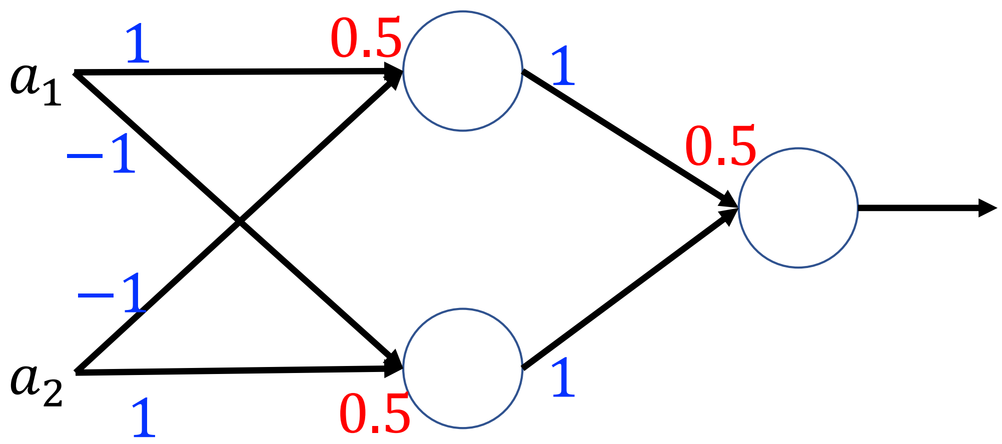
Multi-Layer Perceptron

Hidden Neurons

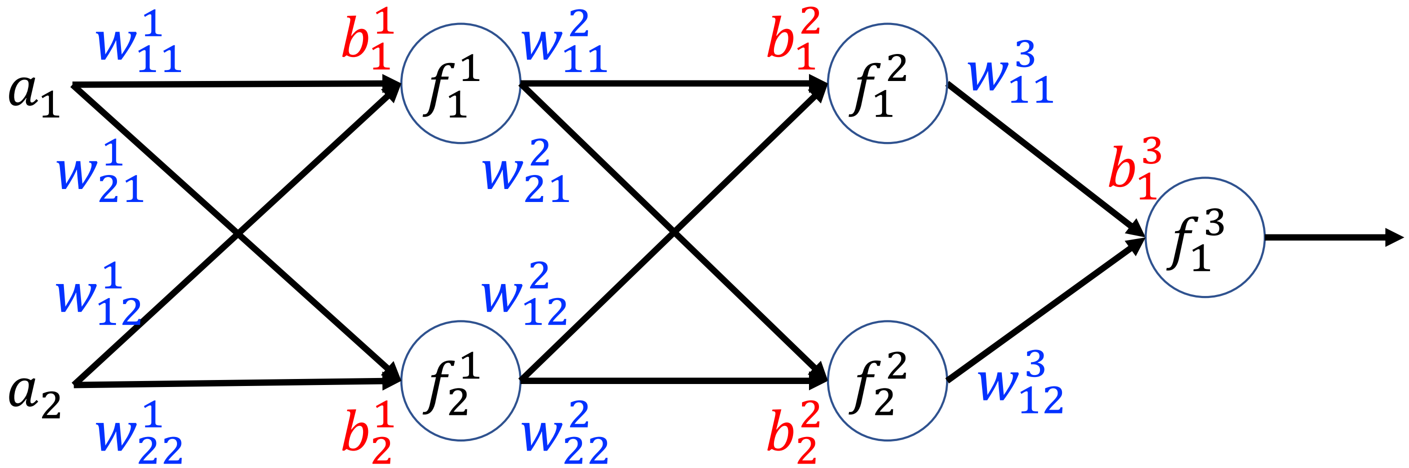
a_1	a_2	"XOR"
1	1	0
0	1	1
1	0	1
0	0	0



Minsky and Papert showed that **in the case of any non-linearly separable problem, such as XOR, in the network architecture *there must be "hidden neurons"*,** *i.e*. the neurons with output not available to the outside world, in order to help turn the problem into a linearly separable one for the outputs.



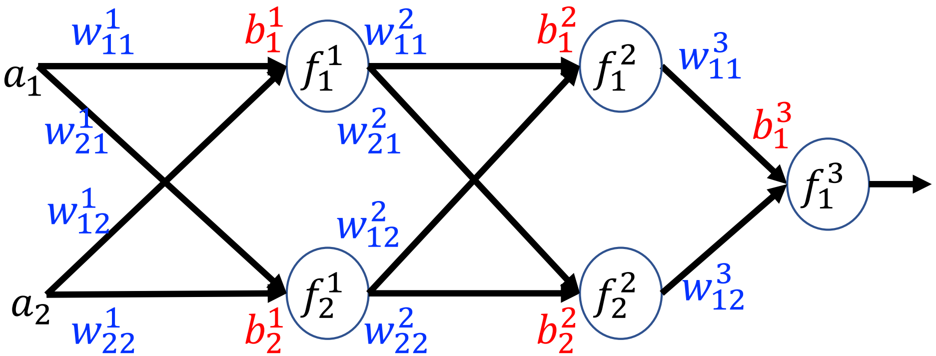
Multilayer Perceptron



A multilayer perceptron (**MLP**) is a layered architecture of neurons, where

- all the neurons are divided into l subsets, each set is called a layer;
- adjacent layers are fully collected;
- each connection is associated with a real weight and each neuron is associated with a real bias (we assume **the biases in the network are all zero**);
- inputs are real and outputs are real.

Forward Propagation



l: the number of layers,
n^l: the number of neurons in the l-th layer
$n = n^0$: the number of input neurons (0-th layer).
$m = n^l$: the number of output neurons (l-th layer).
X^l: the output value of the l-th layer.
$a = X^0$: the input value of the MLP.
$X = X^l$: the output value of the MLP.
$f^l:\mathbb{R}^{n_l} \to \mathbb{R}^{n_l}$: activation function of the l-th layer

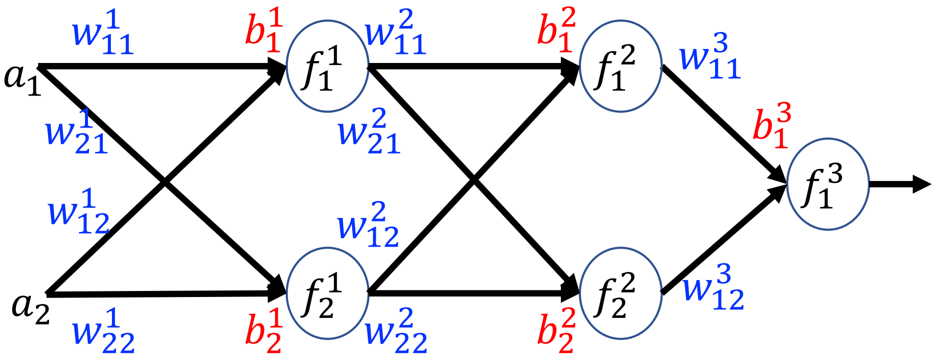
We first consider the first hidden layer. The output of the j-th neuron in the first layer is

$$X_j^1 = f_j^1(S_j^1) = f_j^1\left(\sum_{i=0}^{n^0} w_{ji}^1 X_i^0 + \underbrace{b_j^1}_{=0}\right) = F_j^1(w_j^1, X^0), \qquad j = 1, \cdots, n^1$$

We can then describe the above relation in a compact form:

$$X^1 = F^1(w^1, X^0)$$

Forward Propagation



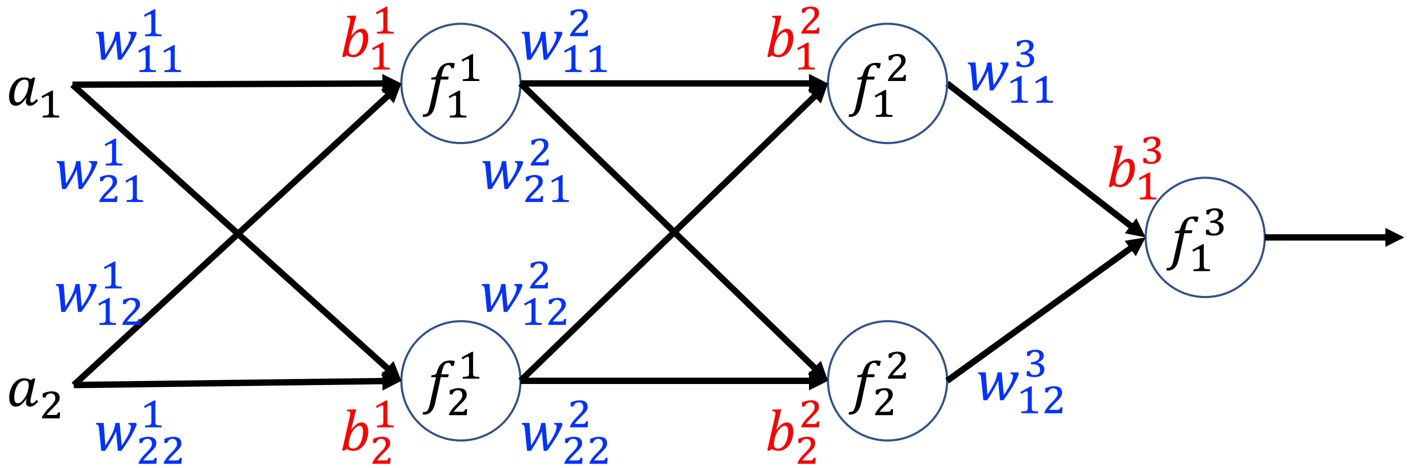
l: the number of layers,
n^l: the number of neurons in the l-th layer
$n = n^0$: the number of input neurons (0-th layer).
$m = n^l$: the number of output neurons (l-th layer).
X^l: the output value of the l-th layer.
$a = X^0$: the input value of the MLP.
$X = X^l$: the output value of the MLP.
f^l:$\mathbb{R}^{n_l} \to \mathbb{R}^{n_l}$: activation function of the l-th layer

Similarly, we can derive the relation for the following layers:

$$X^1 = F^1(w^1, X^0)$$
$$X^2 = F^2(w^2, X^1)$$
$$X^3 = F^3(w^3, X^2)$$
$$\cdots$$
$$X^l = F^l\left(w^l, X^{l-1}\right)$$

The process of such layer-by-layer calculation to obtain the output of a multilayer perceptron is thus called forward propagation.

Multilayer Perceptron



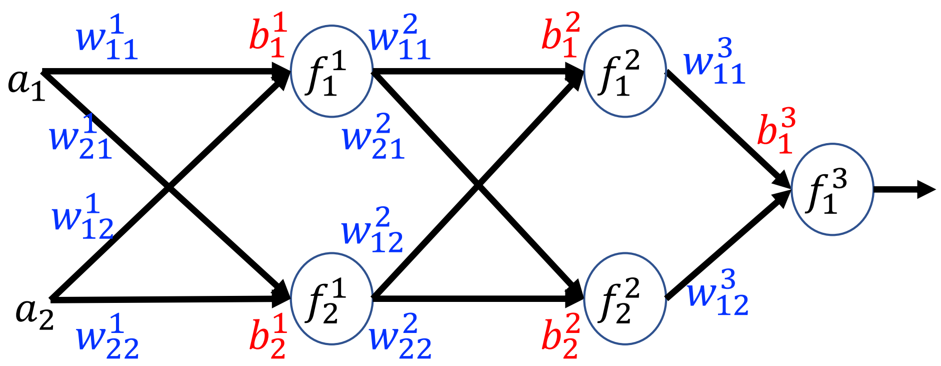
- Issue: The hidden neurons cannot be trained by causing their outputs to become closer to the desired values given by the training set.
- As Rosenblatt himself noted, the multilayer feedforward networks posed the ***structural credit assignment problem***:

when **an error** is made at the output of a network, how is credit (or blame) to be assigned to neurons deep within the network?

Topic of Today's Lecture

Gradient Decent and
Activation Function Design

Error Function for a Single Input

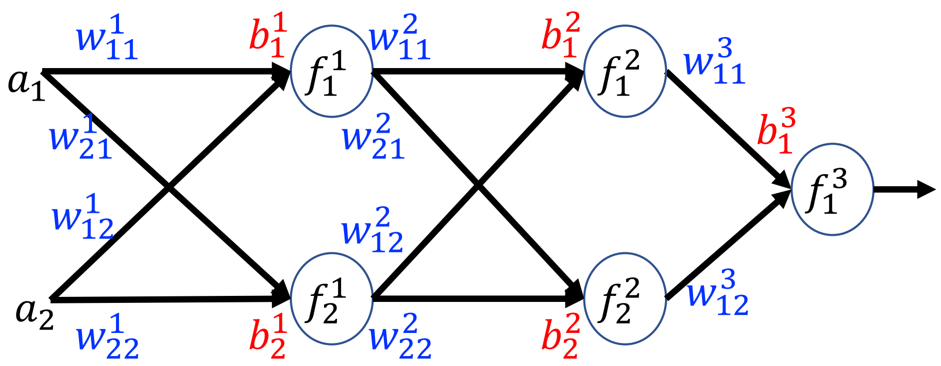


l: the number of layers,
n^l: the number of neurons in the l-th layer
$n = n^0$: the number of input neurons (0-th layer).
$m = n^l$: the number of output neurons (l-th layer).
X^l: the output value of the l-th layer.
$a = X^0$: the input value of the MLP.
$X = X^l$: the output value of the MLP.
f^l:$\mathbb{R}^{n_l} \rightarrow \mathbb{R}^{n_l}$: activation function of the l-th layer

The output error of a multilayer perceptron for the k-th input pattern is a half of the squared error:

$$E^k = \frac{1}{2}\sum_{j=1}^{m}\left(e_j^k\right)^2 = \frac{1}{2}\sum_{j=1}^{m}\left(t_j^k - X_j^k\right)^2,$$

Error Function for a Single Input

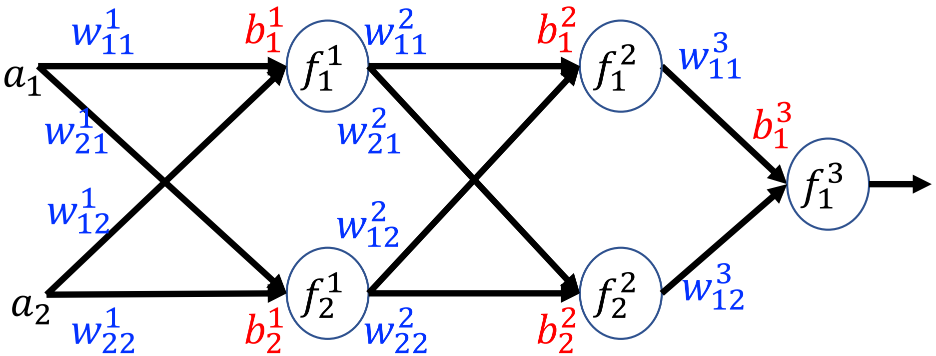


l: the number of layers,
n^l: the number of neurons in the l-th layer
$n = n^0$: the number of input neurons (0-th layer).
$m = n^l$: the number of output neurons (l-th layer).
X^l: the output value of the l-th layer.
$a = X^0$: the input value of the MLP.
$X = X^l$: the output value of the MLP.
$f^l:\mathbb{R}^{n_l} \to \mathbb{R}^{n_l}$: activation function of the l-th layer

Now we define the performance of multilayer perceptron over a Data set D as a half of the total squared error:

$$E = \sum_{k=1}^{r} E^k = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(e_j^k\right)^2 = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - X_j^k\right)^2,$$

Multilayer Perceptron

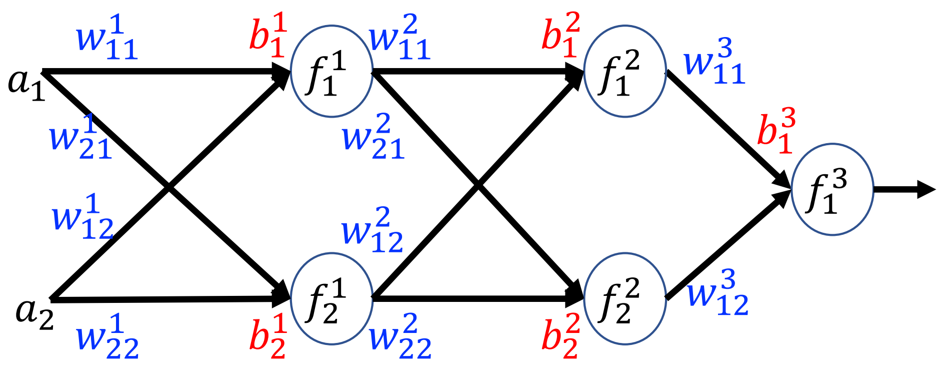


Now we define the performance of multilayer perceptron over a Data set D as a half of the total squared error:

$$E = \sum_{k=1}^{r} E^k = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(e_j^k\right)^2$$

$$= \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - X_j^k\right)^2,$$

Multilayer Perceptron



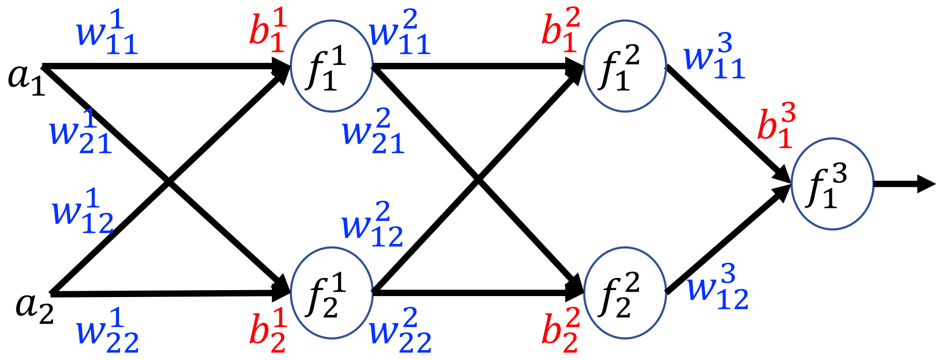
Now we define the performance of multilayer perceptron over a Data set D as a half of the total squared error:

$$E = \sum_{k=1}^{r} E^k = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(e_j^k\right)^2$$
$$= \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - X_j^k\right)^2,$$

- During the training stage, the input value a to the network are the input vectors of the training set, which are *constant parameters* for the process.
- Therefore, the **MLP error function** E **is a function of the weights of connections only:**

$$E = \sum_{k=1}^{r} E^k = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(e_j^k\right)^2 = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - F_j\left(w^l, w^{l-1}, \cdots, w^1, a^k\right)\right)^2 = E(W)$$

Multilayer Perceptron

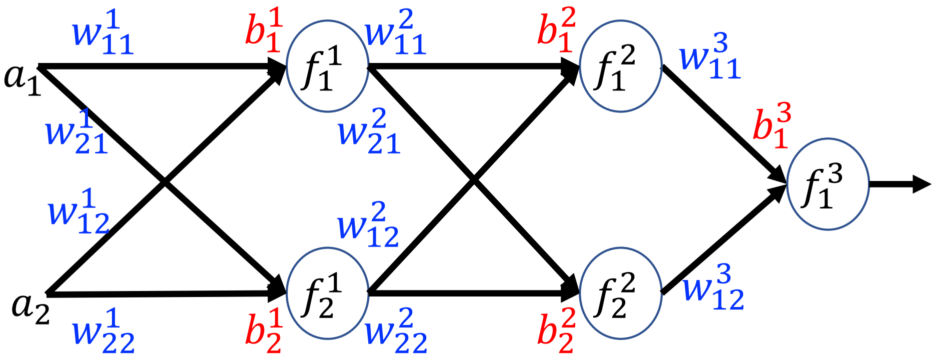


Now we define the performance of multilayer perceptron over a Data set D as a half of the total squared error:

$$E = \sum_{k=1}^{r} E^k = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(e_j^k\right)^2$$
$$= \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - X_j^k\right)^2,$$

- During the training stage, the input value a to the network are the input vectors of the training set, which are *constant parameters* for the process.
- Therefore, the **MLP error function** E **is a function of the weights of connections only:**
  $$E = \sum_{k=1}^{r} E^k = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(e_j^k\right)^2 = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - F_j\left(w^l, w^{l-1}, \cdots, w^1, a^k\right)\right)^2 = E(W)$$
- The better the MLP performs, the smaller the MLP error function E is.

Multilayer Perceptron

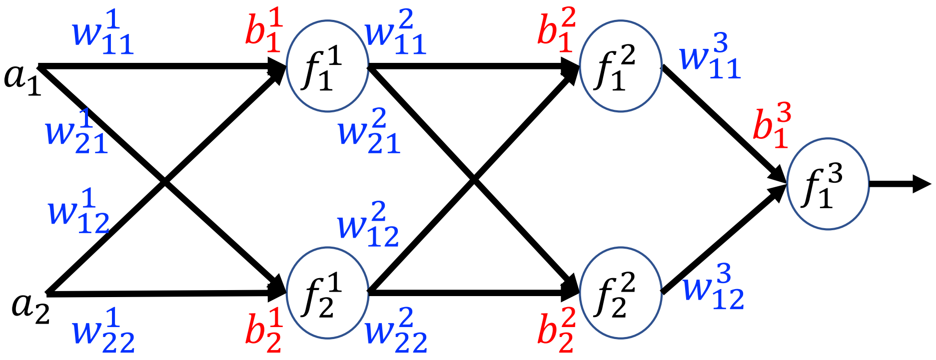


Now we define the performance of multilayer perceptron over a Data set D as a half of the total squared error:

$$E = \sum_{k=1}^{r} E^k = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(e_j^k\right)^2$$
$$= \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - X_j^k\right)^2,$$

- During the training stage, the input value a to the network are the input vectors of the training set, which are *constant parameters* for the process.
- Therefore, the **MLP error function E is a function of the weights of connections only:**
$$E = \sum_{k=1}^{r} E^k = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(e_j^k\right)^2 = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - F_j\left(w^l, w^{l-1}, \cdots, w^1, a^k\right)\right)^2 = E(W)$$
- <u>The better the MLP performs, the smaller the MLP error function E is.</u>
- Thus MLP learning can be considered as the optimization problem: <u>$\min_W E(w)$</u>

Multilayer Perceptron



Now we define the performance of multilayer perceptron over a Data set D as a half of the total squared error:

$$E = \sum_{k=1}^{r} E^k = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(e_j^k\right)^2$$
$$= \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - X_j^k\right)^2,$$

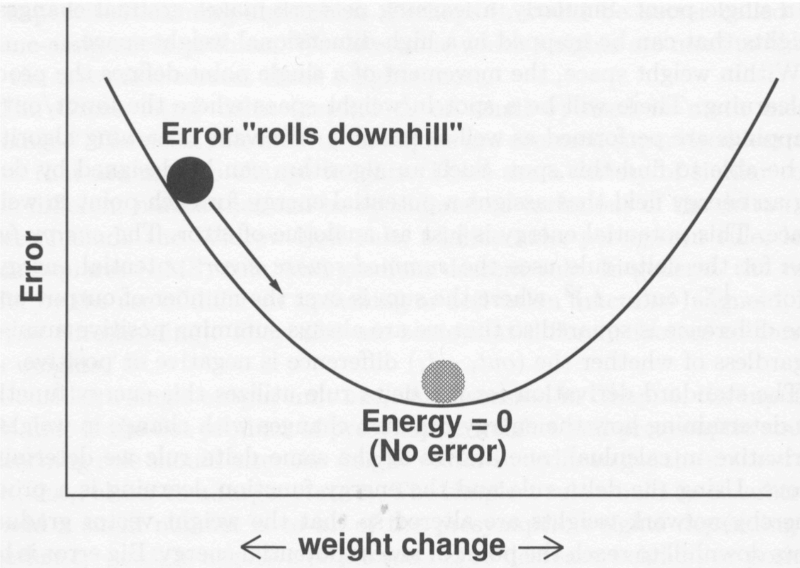
- During the training stage, the input value a to the network are the input vectors of the training set, which are *constant parameters* for the process.
- Therefore, the **MLP error function** E **is a function of the weights of connections only:**

$$E = \sum_{k=1}^{r} E^k = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(e_j^k\right)^2 = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - F_j\left(w^l, w^{l-1}, \cdots, w^1, a^k\right)\right)^2 = E(W)$$

- The better the MLP performs, the smaller the MLP error function E is.
- Thus MLP learning can be considered as the optimization problem: $\min_W E(w)$

Gradient Decent!

Gradient Decent

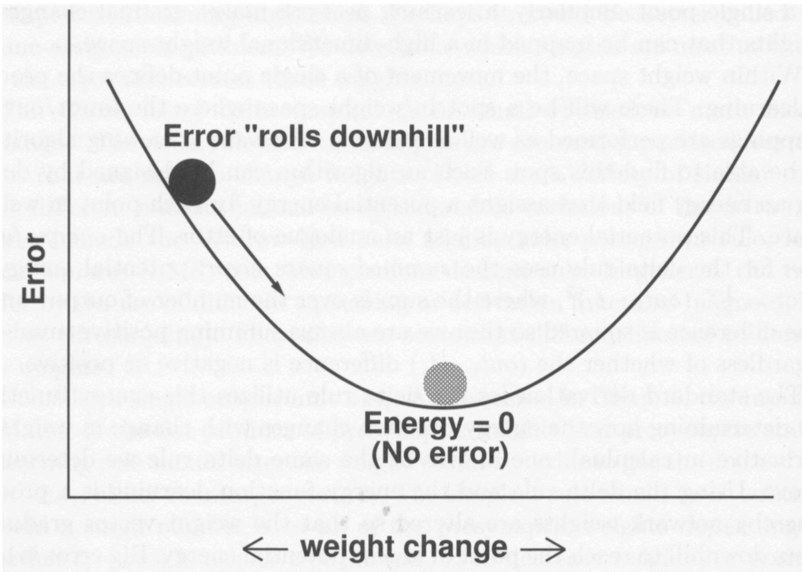


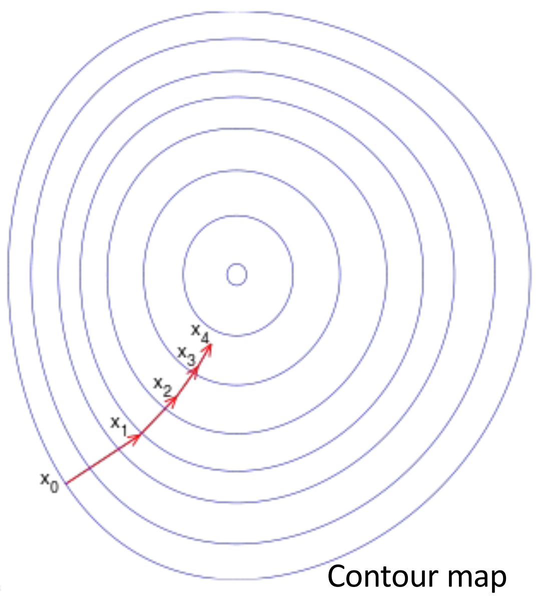
Gradient descent is based on the observation that if the multi-variable function $F(x)$ is defined and differentiable in a neighborhood of a point a, then $F(x)$ decreases fastest if one goes from a in the direction of the negative gradient of F at a, $-\nabla F(a)$. It follows that, if

$$a' = a + \gamma\big(-\nabla F(a)\big) = a - \gamma \nabla F(a)$$

For a $\gamma \in \mathbb{R}_+$ small enough, then $F(a) \geq F(a')$

Gradient Decent





Contour map

With this observation, one starts with an initial guess x_0 for a local minimum of F, and considers the sequence $x_0, x_1, x_2, \cdots$ such that

$$x_{n+1} = x_n - \gamma \ \nabla F(x_n).$$

We can have monotonic sequence

$$F(x_0) \geq F(x_1) \geq F(x_2) \geq \cdots$$

So hopefully, the sequence $\{x_n\}$ converges to the desired local minimum x': $\nabla F(x') = 0$.

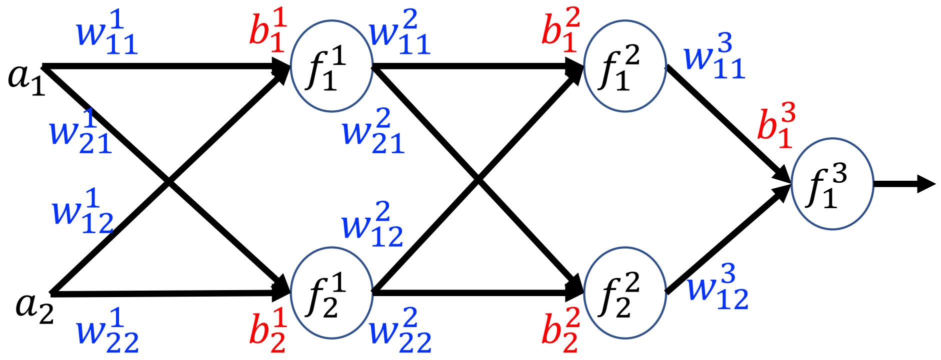
Source: Wikipedia.org

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

$$a_{n+1} = a_n - \gamma \nabla F(a_n)$$

For a $\gamma \in \mathbb{R}_+$ small enough, then $F(a_n) \geq F(a_{n+1})$

Learning of a Multilayer Perceptron



The output error function E^k for the k-th input pattern is:

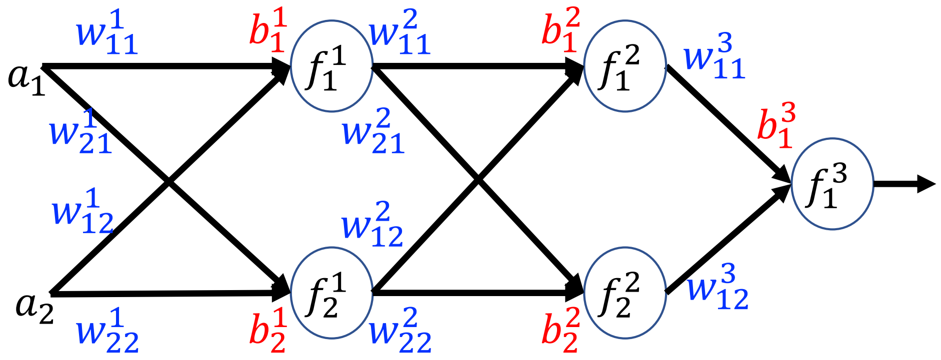
$$E^k = \frac{1}{2}\sum_{j=1}^{m}\left(t_j^k - X_j^k\right)^2,$$

The **MLP error function** E **is :**

$$E = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - F_j\left(w^l, w^{l-1}, \cdots, w^1, a^k\right)\right)^2$$

- Gradient descent method addresses the issue of how to update weights.

Learning of a Multilayer Perceptron



The output error function E^k for the k-th input pattern is:

$$E^k = \frac{1}{2}\sum_{j=1}^{m}\left(t_j^k - X_j^k\right)^2,$$

The **MLP error function** E **is :**

$$E = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - F_j\left(w^l, w^{l-1}, \cdots, w^1, a^k\right)\right)^2$$

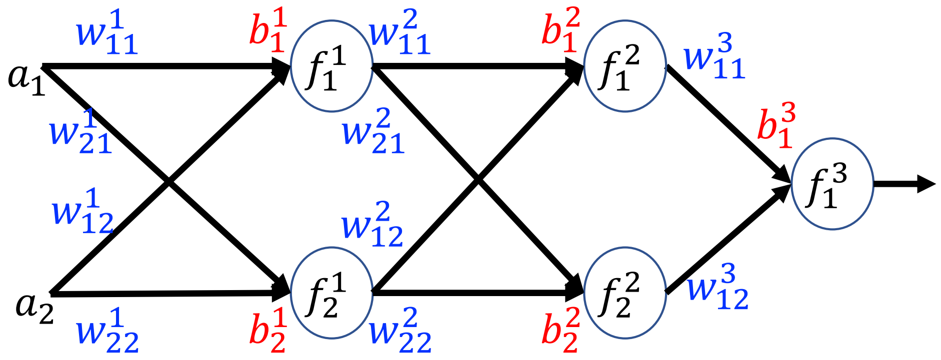
One of the most popular techniques is called

error backpropagation,

where the error of output neurons is propagated back to derive the weight adjustment of a given hidden neuron, based on how much the neuron contributes to the output error.

The ***backpropagation*** algorithm updates the weights of connections w computationally efficiently based on the **method of *gradient descent.***

Learning of a Multilayer Perceptron



The output error function E^k for the k-th input pattern is:

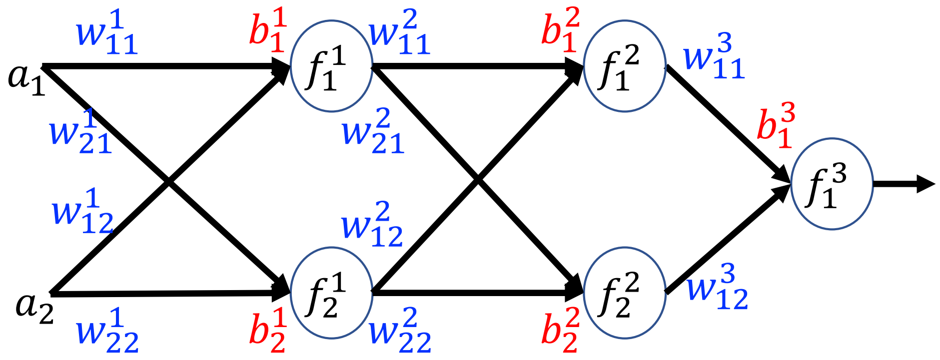
$$E^k = \frac{1}{2}\sum_{j=1}^{m}\left(t_j^k - X_j^k\right)^2,$$

The **MLP error function** E **is :**

$$E = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - F_j\left(w^l, w^{l-1}, \cdots, w^1, a^k\right)\right)^2$$

- Gradient descent method addresses the issue of how to update weights.
- The ***backpropagation*** algorithm makes the weight updating efficient.

Learning of a Multilayer Perceptron



The output error function E^k for the k-th input pattern is:

$$E^k = \frac{1}{2}\sum_{j=1}^{m}\left(t_j^k - X_j^k\right)^2,$$

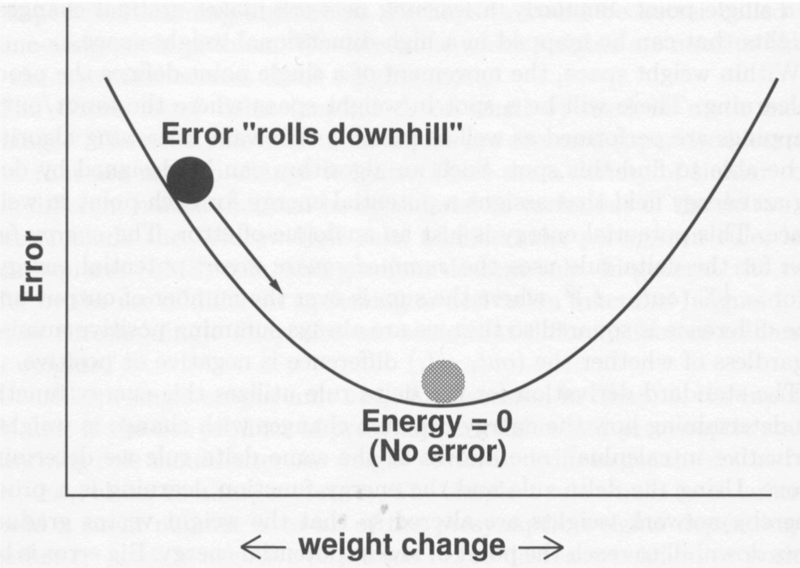
The **MLP error function** E **is :**

$$E = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - F_j\left(w^l, w^{l-1}, \cdots, w^1, a^k\right)\right)^2$$

- Gradient descent method addresses the issue of how to update weights.
- The ***backpropagation*** algorithm makes the weight updating efficient.

We start from gradient descent method for MLP.

Learning of a Multilayer Perceptron



The **MLP error function** E **is :**

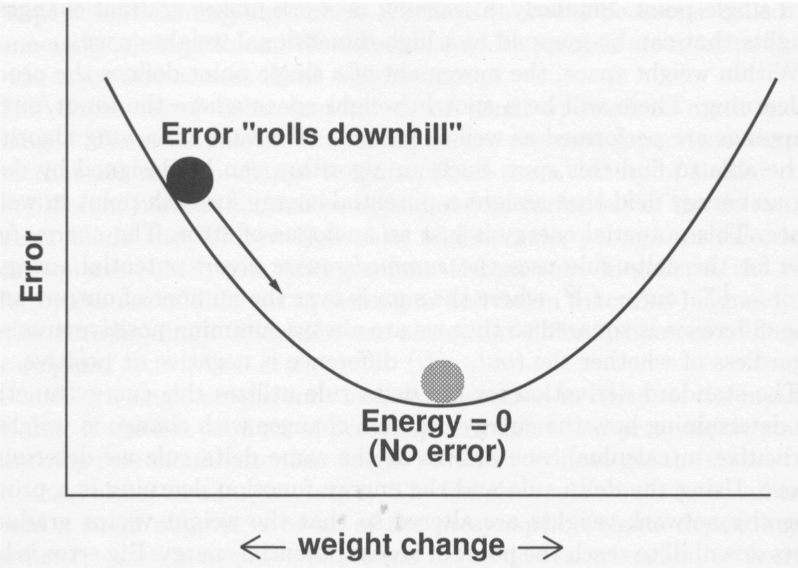
$$E = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - F_j\left(w^l, w^{l-1}, \cdots, w^1, a^k\right)\right)^2$$

The ***backpropagation*** algorithm looks for the minimum of the error function E in the space of weights of connections w using the **method of *gradient descent.***

The gradient the multi-variate function E is defined as:

$$\nabla E = \left(\frac{\partial E}{\partial w_{11}^1}, \cdots, \frac{\partial E}{\partial w_{n^1 n^0}^1}, \frac{\partial E}{\partial w_{11}^2}, \cdots, \frac{\partial E}{\partial w_{n^2 n^1}^2}, \cdots, \frac{\partial E}{\partial w_{11}^l}, \cdots, \frac{\partial E}{\partial w_{n^l n^{l-1}}^l}\right)$$

Learning of a Multilayer Perceptron



The **MLP error function** E **is :**

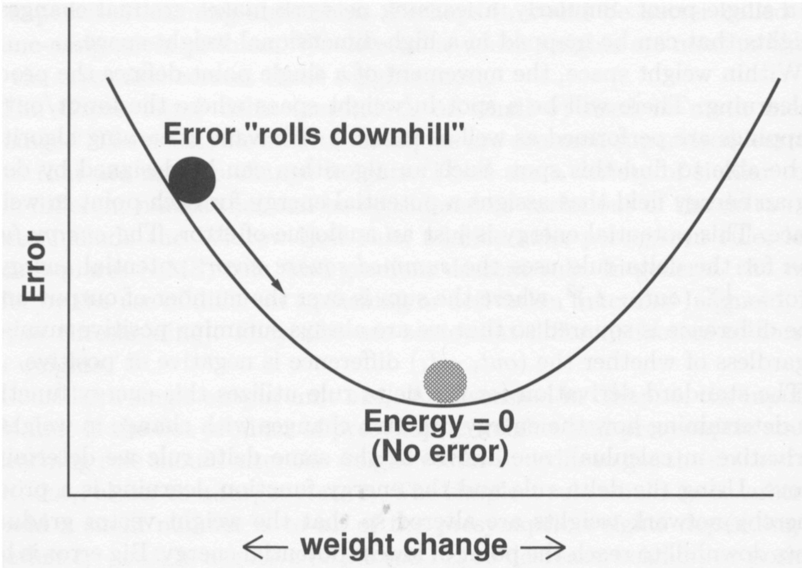
$$E = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - F_j\left(w^l, w^{l-1}, \cdots, w^1, a^k\right)\right)^2$$

The gradient the multi-variate function E is defined as

$$\nabla E = \left(\frac{\partial E}{\partial w_{11}^1}, \cdots, \frac{\partial E}{\partial w_{n^1 n^0}^1}, , \cdots, \frac{\partial E}{\partial w_{11}^l}, \cdots, \frac{\partial E}{\partial w_{n^l n^{l-1}}^l}\right)$$

$\frac{\partial E}{\partial w_{ji}^h}$: is partial derivative of the error function E with respect to the weight of connection between j-th neuron in the layer h and i-th neuron in the previous layer $h-1$.

Learning of a Multilayer Perceptron



The **MLP error function** E **is :**

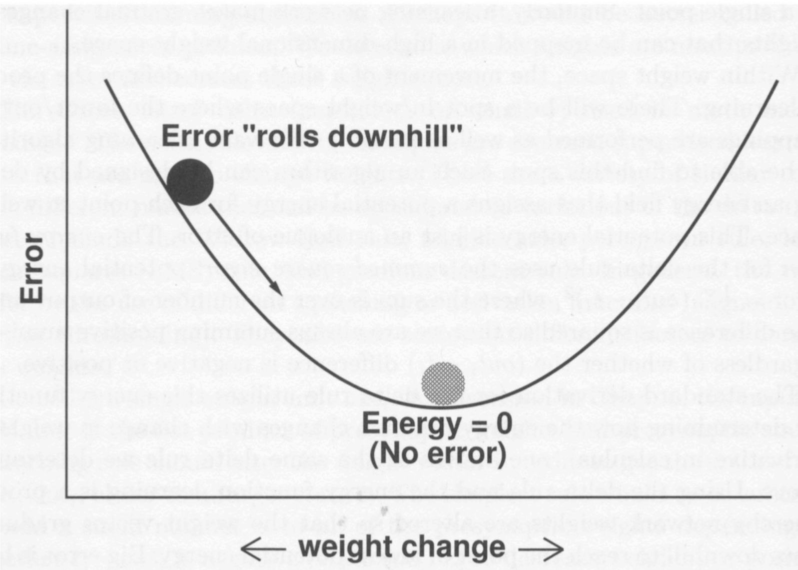
$$E = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - F_j\left(w^l, w^{l-1}, \cdots, w^1, a^k\right)\right)^2$$

The gradient the multi-variate function E is defined as

$$\nabla E = \left(\frac{\partial E}{\partial w_{11}^1}, \cdots, \frac{\partial E}{\partial w_{n^1 n^0}^1}, , \cdots, \frac{\partial E}{\partial w_{11}^l}, \cdots, \frac{\partial E}{\partial w_{n^l n^{l-1}}^l}\right)$$

Vector ∇E is called gradient of the error function E, and it points in the direction along which E increases most rapidly.

Learning of a Multilayer Perceptron



The **MLP error function E is :**

$$E = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - F_j\left(w^l, w^{l-1}, \cdots, w^1, a^k\right)\right)^2$$

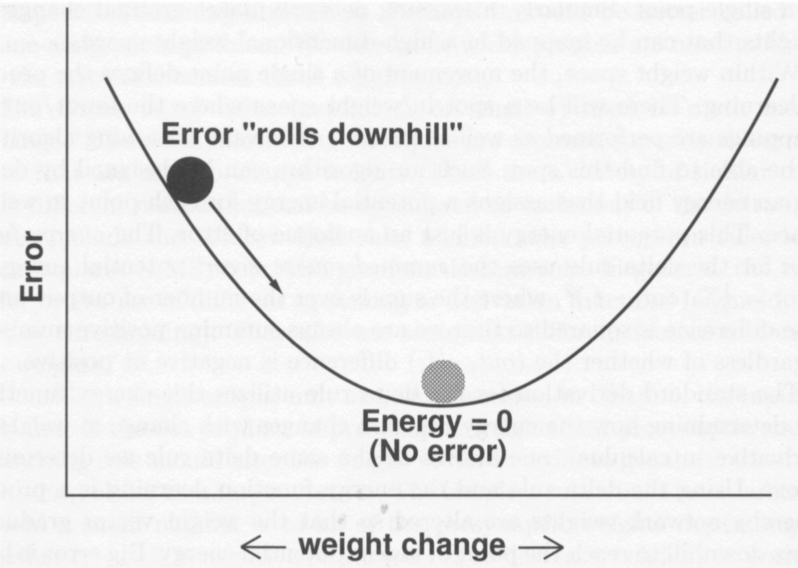
The gradient the multi-variate function E is defined as

$$\nabla E = \left(\frac{\partial E}{\partial w_{11}^1}, \cdots, \frac{\partial E}{\partial w_{n^1 n^0}^1}, , \cdots, \frac{\partial E}{\partial w_{11}^l}, \cdots, \frac{\partial E}{\partial w_{n^l n^{l-1}}^l}\right)$$

Vector ∇E is called gradient of the error function E, and it points in the direction along which E increases most rapidly.

We would like to go in the opposite direction to most rapidly minimize E.

Learning of a Multilayer Perceptron



The **MLP error function** E **is :**

$$E = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - F_j\left(w^l, w^{l-1}, \cdots, w^1, a^k\right)\right)^2$$

The gradient the multi-variate function E is defined as

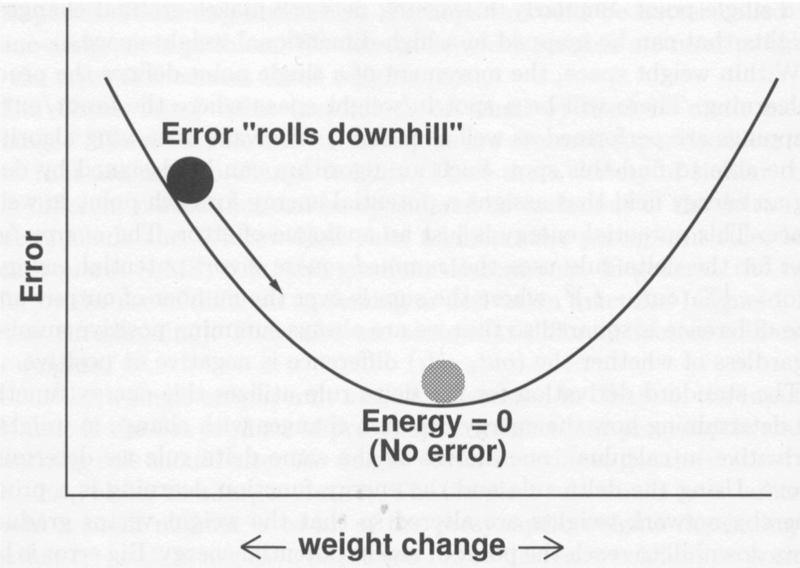
$$\nabla E = \left(\frac{\partial E}{\partial w_{11}^1}, \cdots, \frac{\partial E}{\partial w_{n^1 n^0}^1}, , \cdots, \frac{\partial E}{\partial w_{11}^l}, \cdots, \frac{\partial E}{\partial w_{n^l n^{l-1}}^l}\right)$$

Therefore, during the ***iterative process*** **of** ***gradient descent*** **each weight of connection, including the hidden ones, is updated:**

$$w_{ji}^h = w_{ji}^h + \Delta w_{ji}^h, \text{ where } \Delta w_{ji}^h = -C\frac{\partial E}{\partial w_{ji}^h}$$

Here C represents the learning rate as before.

Learning of a Multilayer Perceptron



The **MLP error function** E **is :**

$$E = \frac{1}{2}\sum\nolimits_{k=1}^{r}\sum\nolimits_{j=1}^{m}\left(t_j^k - F_j\left(w^l, w^{l-1}, \cdots, w^1, a^k\right)\right)^2$$

Its ***gradient:***

$$\nabla E = \left(\frac{\partial E}{\partial w_{11}^1}, \cdots, \frac{\partial E}{\partial w_{n^1 n^0}^1}, , \cdots, \frac{\partial E}{\partial w_{11}^l}, \cdots, \frac{\partial E}{\partial w_{n^l n^{l-1}}^l}\right)$$

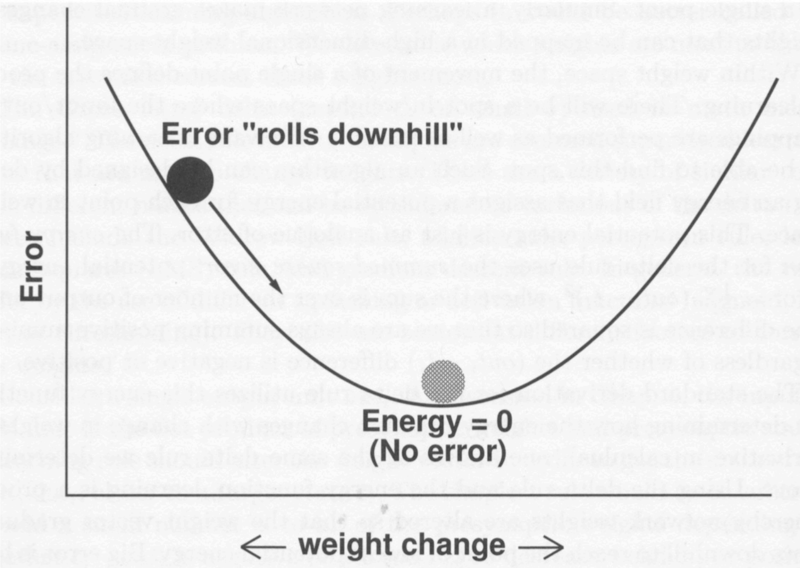
Update rule:

$$w_{ji}^h = w_{ji}^h + \Delta w_{ji}^h, \text{ where } \Delta w_{ji}^h = -C\,\frac{\partial E}{\partial w_{ji}^h}$$

Here C represents the learning rate as before.

Since calculus-based methods of minimization depends on the taking of derivatives, their application to network training requires the error function *E* be a *differentiable* function (almost everywhere).

Learning of a Multilayer Perceptron



Update rule:

$$w_{ji}^{h} = w_{ji}^{h} + \Delta w_{ji}^{h},$$

where $\Delta w_{ji}^{h} = -C\,\frac{\partial E}{\partial w_{ji}^{h}}$

Since calculus-based methods of minimization depends on the taking of derivatives, their application to network training requires the error function *E* be a *differentiable* function.

Recall the **MLP error function** E**:**

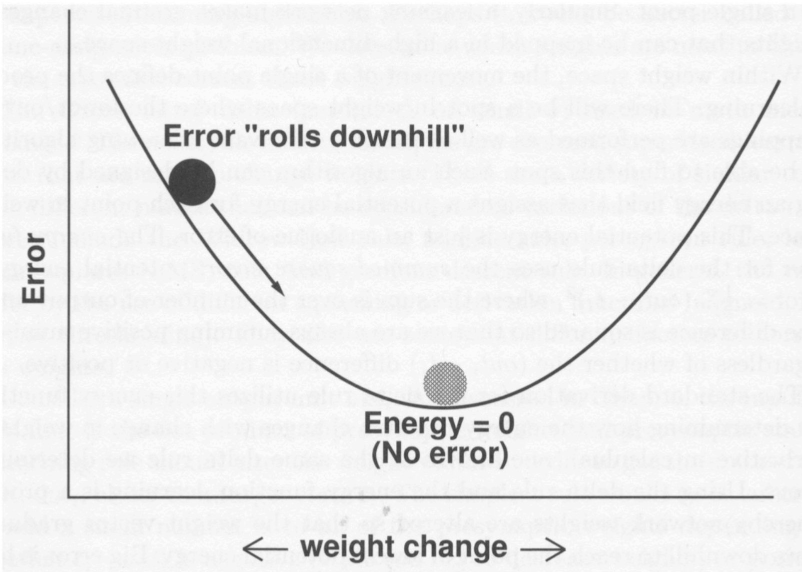
$$E = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - F_j\left(w^l, w^{l-1}, \cdots, w^1, a^k\right)\right)^2$$

$$= \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - f_j^l\left(\sum_{i=0}^{n^{l-1}} w_{ji}^l X_i^{l-1}\right)\right)^2$$

$$= \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - f_j^l\left(\sum_{i=0}^{n^{l-1}} w_{ji}^l \cdot f_i^{l-1}\left(S_i^{l-1}\right)\right)\right)^2 = \cdots$$

$$= \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - f_j^l\left(\cdots f_s^1\left(\sum_{k=0}^{n^0} w_{sk}^1 X_k^0\right)\right)\right)^2$$

Learning of a Multilayer Perceptron



Since calculus-based methods of minimization depends on the taking of derivatives, their application to network training requires the error function *E* be a *differentiable* function.

Recall the **MLP error function** E**:**

$$E = \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - F_j\left(w^l, w^{l-1}, \cdots, w^1, a^k\right)\right)^2$$

$$= \frac{1}{2}\sum_{k=1}^{r}\sum_{j=1}^{m}\left(t_j^k - f_j^l\left(\cdots f_s^1\left(\sum_{k=0}^{n^0} w_{sk}^1 X_k^0\right)\right)\right)^2$$

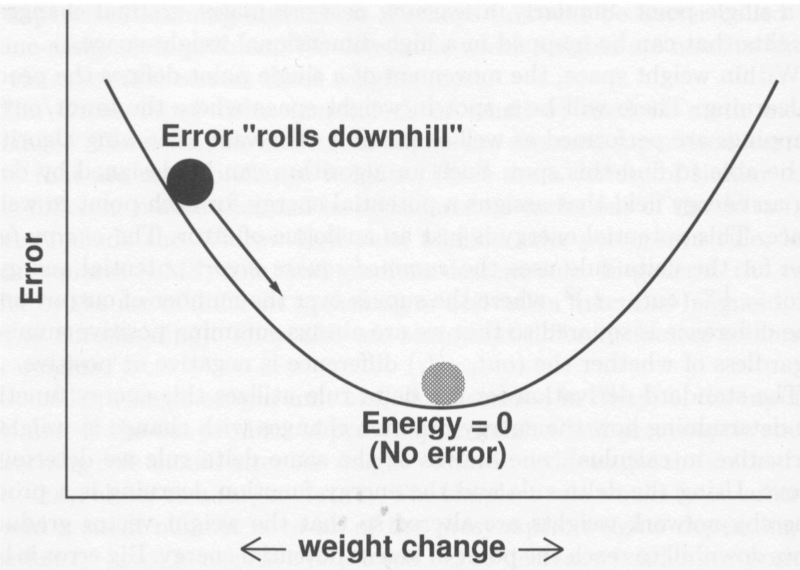
This provides a powerful motivation for using ***continuous and differentiable activation functions*** f.

Update rule:

$$w_{ji}^h = w_{ji}^h + \Delta w_{ji}^h,$$

$$\text{where } \Delta w_{ji}^h = -C\,\frac{\partial E}{\partial w_{ji}^h}$$

Learning of a Multilayer Perceptron



Update rule:

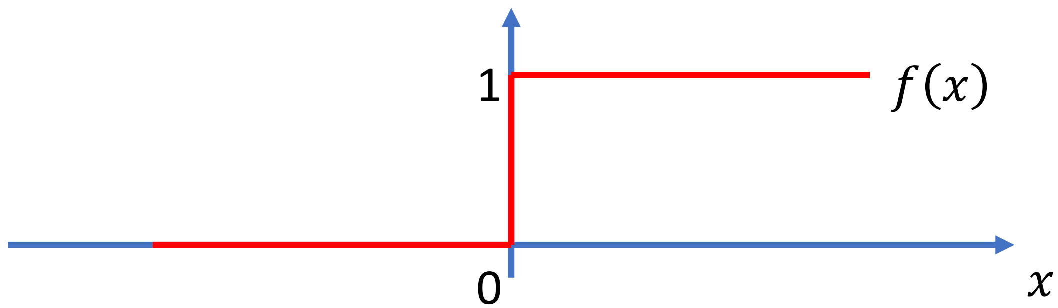
$$w_{ji}^{h} = w_{ji}^{h} + \Delta w_{ji}^{h},$$

$$\text{where } \Delta w_{ji}^{h} = -C\,\frac{\partial E}{\partial w_{ji}^{h}}$$

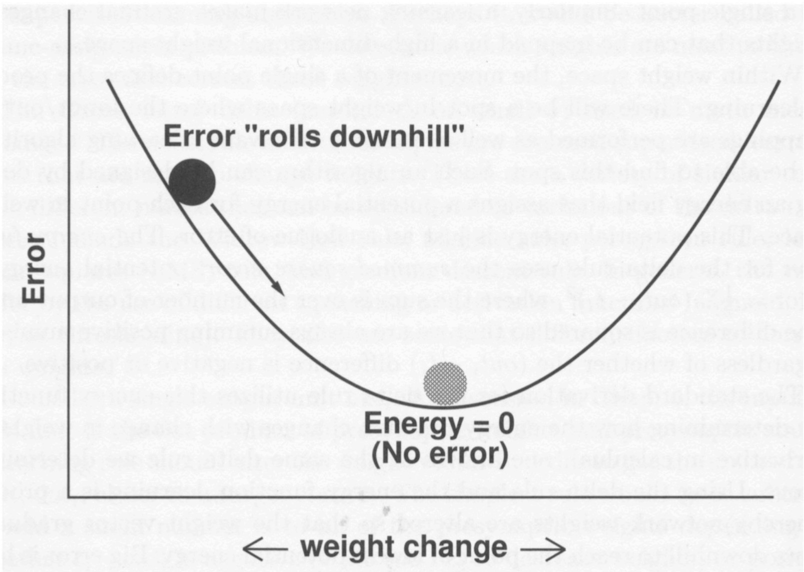
This provides a powerful motivation for using ***continuous and differentiable activation functions f.***

Recall the binary activation function we used for MP neuron and perceptron:

$$f(S) = \begin{cases} 1, & S \geq 0, \\ 0, & S < 0. \end{cases}$$



Learning of a Multilayer Perceptron



Update rule:

$$w_{ji}^{h} = w_{ji}^{h} + \Delta w_{ji}^{h},$$

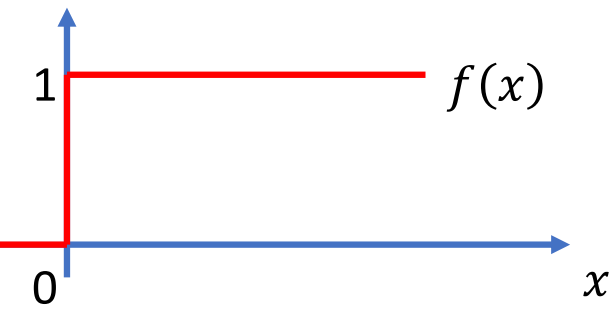
$$\text{where } \Delta w_{ji}^{h} = -C\,\frac{\partial E}{\partial w_{ji}^{h}}$$

This provides a powerful motivation for using ***continuous and differentiable activation functions f.***

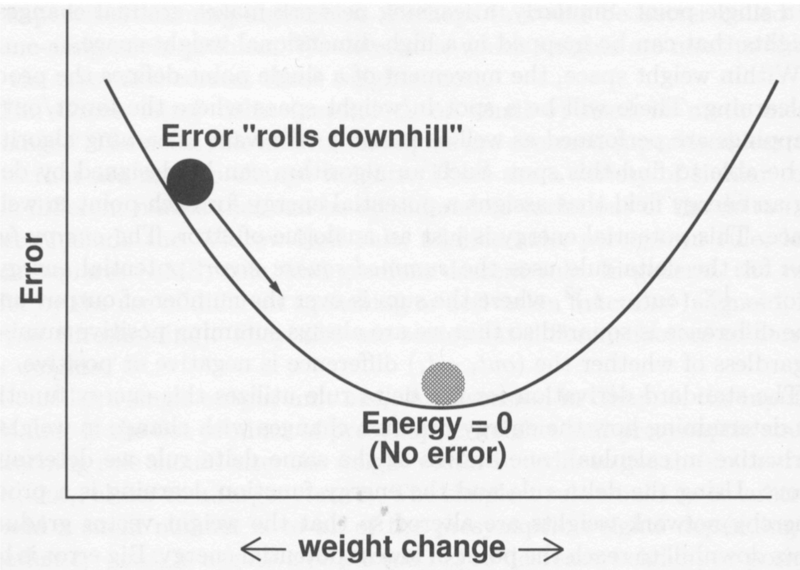
Recall the binary activation function we used for MP neuron and perceptron:

$$f(S) = \begin{cases} 1, & S \geq 0, \\ 0, & S < 0. \end{cases}$$

- **Non-differentiable!**
- **For the differentiable part, the derivative is 0!**



Learning of a Multilayer Perceptron



This provides a powerful motivation for using ***continuous and differentiable activation functions f.***

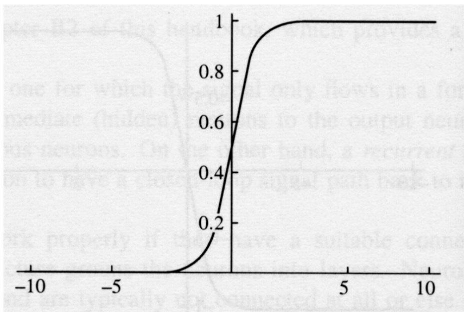
Thus, to make a multiple layer perceptron to be "able to learn" here is a useful generic sigmoidal activation function associated with a hidden or output neuron:

$$f(S) = \frac{\alpha}{1 + e^{-\beta S + \gamma}} + \lambda$$

Update rule:

$$w_{ji}^{h} = w_{ji}^{h} + \Delta w_{ji}^{h},$$

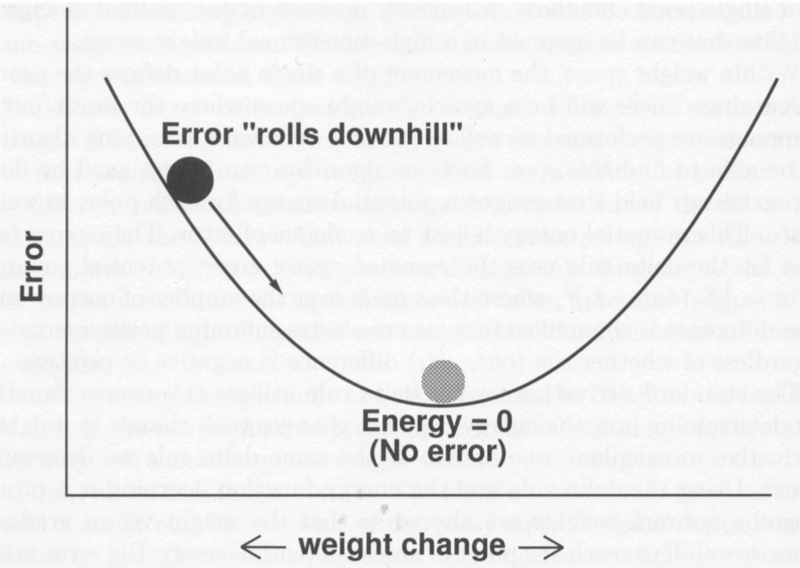
$$\text{where } \Delta w_{ji}^{h} = -C \frac{\partial E}{\partial w_{ji}^{h}}$$



Such sigmoidal function

- has four parameters $\alpha, \beta, \gamma, \lambda$.
- is monotonically increasing.
- has the shape of the s-curve for learning.

Learning of a Multilayer Perceptron



Update rule:

$$w_{ji}^{h} = w_{ji}^{h} + \Delta w_{ji}^{h},$$

$$\text{where } \Delta w_{ji}^{h} = -C\frac{\partial E}{\partial w_{ji}^{h}}$$

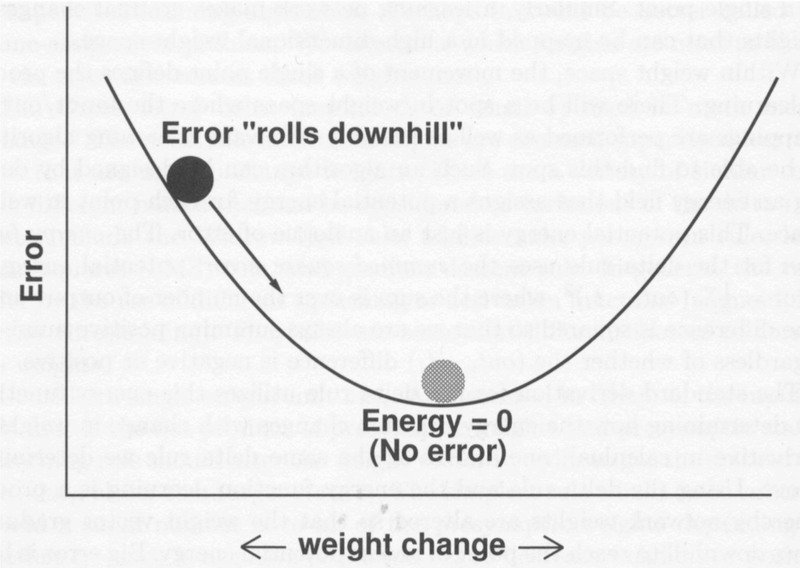
This provides a powerful motivation for using ***continuous and differentiable activation functions f.***

Thus, to make a multiple layer perceptron to be "able to learn" here is a useful generic sigmoidal activation function associated with a hidden or output neuron:

$$f(S) = \frac{\alpha}{1 + e^{-\beta S + \gamma}} + \lambda$$

- **The parameter β has the most significant effect on the slope of this curve: a small value of β corresponds to a gradual curve increase, while its large value corresponds to a steep increase. The case $\beta = \infty$ corresponds to a hard-limiting step function. The product $\alpha\beta$ defines the *steepness* of the curve.**

Learning of a Multilayer Perceptron



This provides a powerful motivation for using ***continuous and differentiable activation functions*** f.

Thus, to make a multiple layer perceptron to be "able to learn" here is a useful generic sigmoidal activation function associated with a hidden or output neuron:

$$f(S) = \frac{\alpha}{1 + e^{-\beta S + \gamma}} + \lambda$$

Update rule:

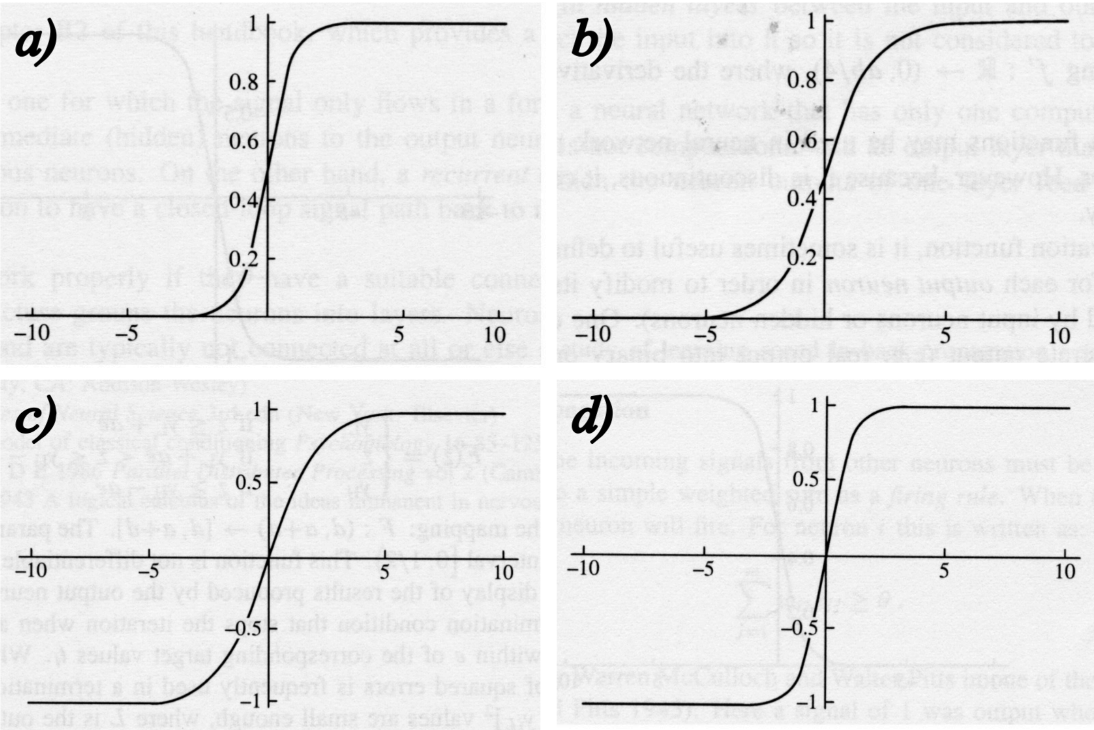
$$w_{ji}^{h} = w_{ji}^{h} + \Delta w_{ji}^{h},$$

$$\text{where } \Delta w_{ji}^{h} = -C \frac{\partial E}{\partial w_{ji}^{h}}$$

- **The product $\alpha\beta$ defines the *steepness* of the curve.**
- **The parameter γ causes a shifting along the horizontal axis and is usually equal to zero.**
- **The parameters α and λ define the range limits for scaling purposes.**

Learning of a Multilayer Perceptron

A few examples of different settings of hyper-parameters



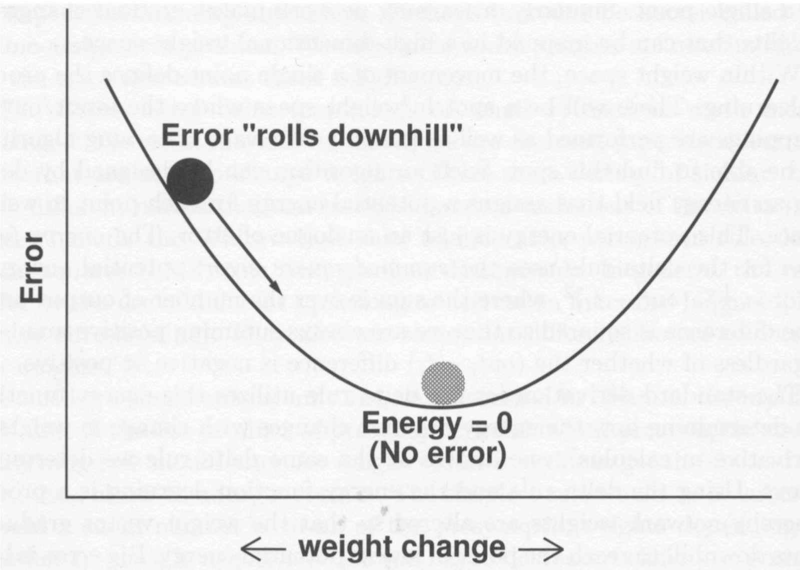
a) Logistic function: $\alpha = 1, \beta = 2, \gamma = 0, \lambda = 0.$

b) **Sigmoid function**: $\alpha = 1, \beta = 1, \gamma = 0, \lambda = 0.$

c) Bipolar function: $\alpha = 2, \beta = 1, \gamma = 0, \lambda = -1.$

d) **Hyperbolic tangent function**: $\alpha = 2, \beta = 2, \gamma = 0, \lambda = -1.$

Learning of a Multilayer Perceptron



This provides a powerful motivation for using ***continuous and differentiable activation functions f.***

Thus, to make a multiple layer perceptron to be "able to learn" here is a useful generic sigmoidal activation function associated with a hidden or output neuron:

$$f(S) = \frac{\alpha}{1 + e^{-\beta S + \gamma}} + \lambda$$

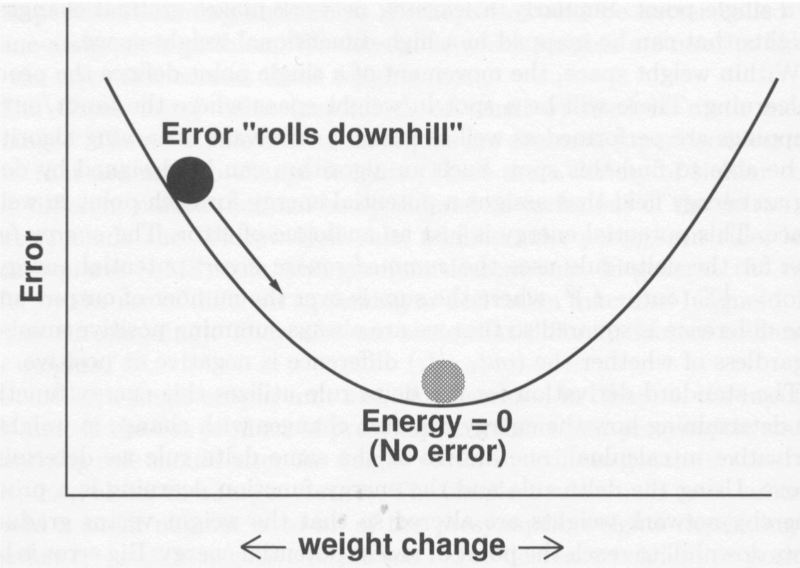
Update rule:

$$w_{ji}^{h} = w_{ji}^{h} + \Delta w_{ji}^{h},$$

$$\text{where } \Delta w_{ji}^{h} = -C \frac{\partial E}{\partial w_{ji}^{h}}$$

Important thing about the generic sigmoid function is that it is differentiable, and it is very easy to compute derivative.

Learning of a Multilayer Perceptron



Update rule:

$$w_{ji}^h = w_{ji}^h + \Delta w_{ji}^h,$$

$$\text{where } \Delta w_{ji}^h = -C\,\frac{\partial E}{\partial w_{ji}^h}$$

This provides a powerful motivation for using ***continuous and differentiable activation functions f.***

Generic sigmoidal activation function :

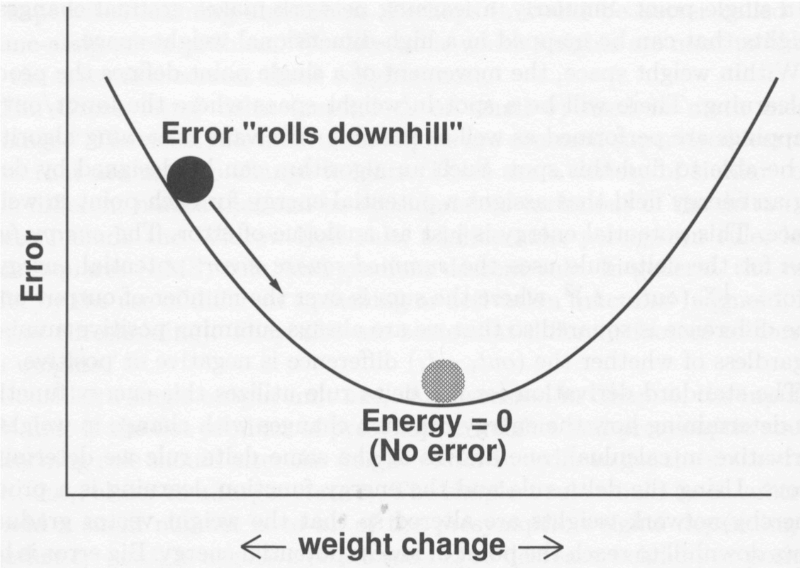
$$f(S) = \frac{\alpha}{1 + e^{-\beta S + \gamma}} + \lambda$$

Its derivative is:

$$f'(S) = \frac{df}{dS} = \frac{\beta}{\alpha} \cdot (f(S) + \lambda)\big(\alpha + \lambda - f(S)\big)$$

It is straight forward to compute the derivative at any particular value of variable S without actual differentiation, if the value of activation function itself is known for that value of S.

Learning of a Multilayer Perceptron



Update rule:

$$w_{ji}^{h} = w_{ji}^{h} + \Delta w_{ji}^{h},$$

$$\text{where } \Delta w_{ji}^{h} = -C\,\frac{\partial E}{\partial w_{ji}^{h}}$$

This provides a powerful motivation for using ***continuous and differentiable activation functions f.***

Generic sigmoidal activation function :

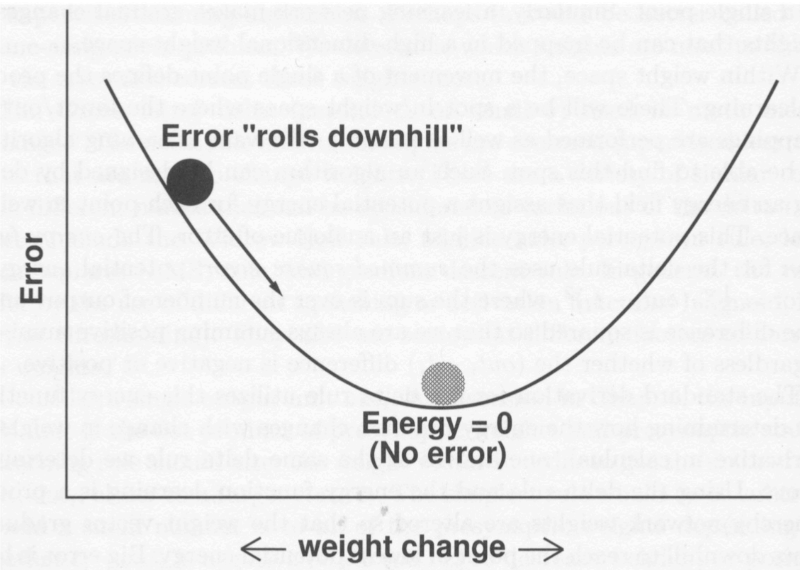
$$f(S) = \frac{\alpha}{1 + e^{-\beta S + \gamma}} + \lambda$$

Its derivative is:

$$f'(S) = \frac{df}{dS} = \frac{\beta}{\alpha} \cdot \left(f(S) + \lambda\right)\left(\alpha + \lambda - f(S)\right)$$

This feature of the sigmoid function is used for *back propagation of corrections to weights of hidden neurons* during multiple layer perceptron training process.

Learning of a Multilayer Perceptron



This provides a powerful motivation for using ***continuous and differentiable activation functions f.***

Generic sigmoidal activation function :

$$f(S) = \frac{\alpha}{1 + e^{-\beta S + \gamma}} + \lambda$$

Its derivative is:

$$f'(S) = \frac{df}{dS} = \frac{\beta}{\alpha} \cdot (f(S) + \lambda)\big(\alpha + \lambda - f(S)\big)$$

Update rule:

$$w_{ji}^h = w_{ji}^h + \Delta w_{ji}^h,$$

where $\Delta w_{ji}^h = -C \frac{\partial E}{\partial w_{ji}^h}$

If all activation functions $f(S)$ in the network are differentiable, then we can the *chain rule* to calculate the partial derivative of the error function E with respect to the weight of a specific connection.

Please review the chain rule before the next lecture.