Back Propagation Algorithm

K Kotecha

Introduction

- As we have seen, a perceptron can only express a linear decision surface.
- A multi-layer NN learned by the back-propagation (BP) algorithm can represent *highly non-linear decision surfaces*
- The BP learning algorithm is used to learn the weights of a multi-layer NN *Fixed structure (i.e., fixed set of neurons and interconnections)*
- For every neuron the activation function must be *continuously differentiable*
- The BP algorithm *employs gradient descent in the weight* update rule
- To minimize the error between the actual output values and the desired output ones, given the training instances

Back Propagation Algorithm

- Back-propagation algorithm searches for the weights vector that minimizes the total error made over the training set
- Back-propagation consists of the two phases

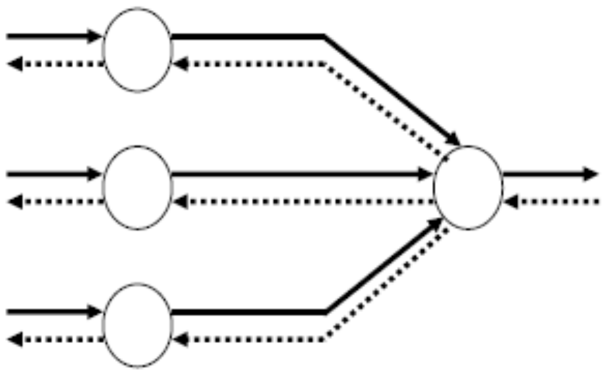
1. Signal forward phase.

   The input signals (i.e., the input vector) are propagated (forwards) from the input layer to the output layer (through the hidden layers)

2. Error backward phase

- Since the desired output value for the current input vector is known, the error is computed
- Starting at the output layer, the error is propagated backwards through the network, layer by layer, to the input layer
- The error back-propagation is performed by recursively computing the local gradient of each neuron

Forward vs backward phase



Signal forward phase
• Network activation

Error backward phase
•Output error computation
•Error propagation

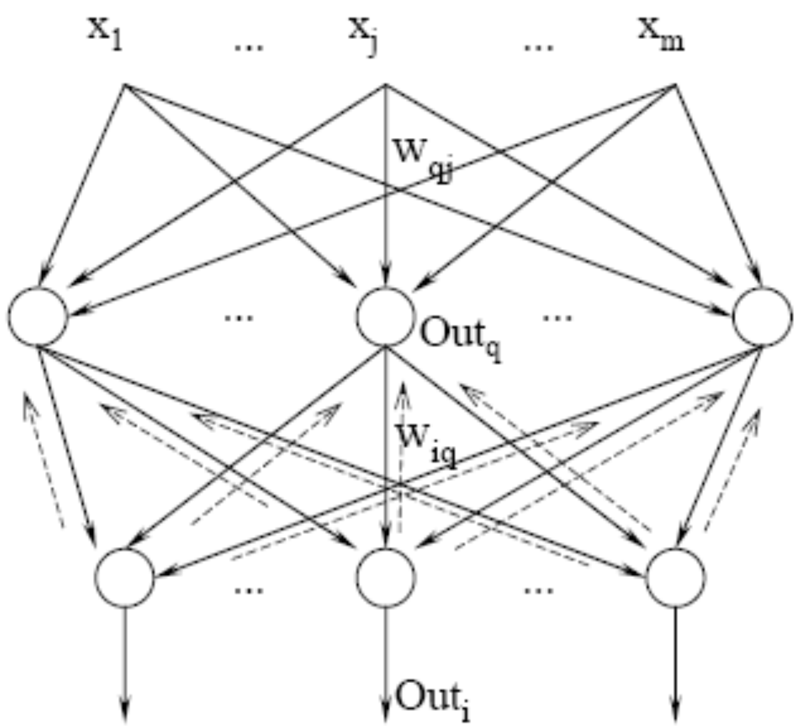
The Network- Derivation

- Let's use this 3-layer NN to illustrate the details of the BP learning algorithm
- *m* input signals x_j (j=1..m)
- *l* hidden neurons z_q (q=1..l)
- *n* output neurons y_i (i=1..n)
- w_{qj} is the weight of the interconnection from input signal x_j to hidden neuron z_q
- w_{iq} is the weight of the interconnection from hidden neuron z_q to output neuron y_i
- Out_q is the (local) output value of hidden neuron z_q
- Out_i is the network output w.r.t. the output neuron y_i

Input x_j (j=1..m)

Hidden neuron z_q (q=1..l)

Output neuron y_i (i=1..n)



Forward Phase

- For each training instance ***x***
  - The input vector ***x*** is *propagated* from the input layer to the output layer
  - The network produces an actual output ***Out*** (i.e., a vector of Out_i, $i=1..n$)

- Given an input vector ***x***, a neuron z_q in the hidden layer receives a net input of

$$Net_q = \sum_{j=1}^{m} w_{qj} x_j$$

…and produces a (local) output of

$$Out_q = f(Net_q) = f\left(\sum_{j=1}^{m} w_{qj} x_j\right)$$

where $f(.)$ is the activation (transfer) function of neuron z_q

Forward Phase

- The net input for a neuron y_i in the output layer is

$$Net_i = \sum_{q=1}^{l} w_{iq} Out_q = \sum_{q=1}^{l} w_{iq} f\left(\sum_{j=1}^{m} w_{qj} x_j \right)$$

- Neuron y_i produces the output value (i.e., an output of the network)

$$Out_i = f(Net_i) = f\left(\sum_{q=1}^{l} w_{iq} Out_q \right) = f\left(\sum_{q=1}^{l} w_{iq} f\left(\sum_{j=1}^{m} w_{qj} x_j \right) \right)$$

- The vector of output values Out_i (i=1..n) is the actual network output, given the input vector $\boldsymbol{x}$

Backward Phase

- For each training instance ***x***
  - The error signals resulting from the difference between the desired output ***d*** and the actual output ***Out*** are computed
  - The error signals are *back-propagated* from the output layer to the previous layers to update the weights
- Before discussing the error signals and their back propagation, we first define an error (cost) function

$$E(w) = \frac{1}{2}\sum_{i=1}^{n}(d_i - Out_i)^2 = \frac{1}{2}\sum_{i=1}^{n}\left[d_i - f(Net_i)\right]^2$$

$$= \frac{1}{2}\sum_{i=1}^{n}\left[d_i - f\left(\sum_{q=1}^{l} w_{iq} Out_q\right)\right]^2$$

Backward Phase

- According to the gradient-descent method, the weights in the **hidden-to-output** connections are updated by

$$\Delta w_{iq} = -\eta \frac{\partial E}{\partial w_{iq}}$$

- Using the derivative chain rule for $\partial E/\partial w_{iq}$, we have

$$\Delta w_{iq} = -\eta \left[\frac{\partial E}{\partial Out_i}\right]\left[\frac{\partial Out_i}{\partial Net_i}\right]\left[\frac{\partial Net_i}{\partial w_{iq}}\right] = \eta\left[d_i - Out_i\right]\left[f'(Net_i)\right]\left[Out_q\right] = \eta\delta_i Out_q$$

(note that the negative sign is incorporated in $\partial E/\partial Out_i$)

- $\boldsymbol{\delta_i}$ is the **error signal** of neuron y_i in the **output layer**

$$\delta_i = -\frac{\partial E}{\partial Net_i} = -\left[\frac{\partial E}{\partial Out_i}\right]\left[\frac{\partial Out_i}{\partial Net_i}\right] = \left[d_i - Out_i\right]\left[f'(Net_i)\right]$$

where Net_i is the net input to neuron y_i in the output layer, and $f'(\mathrm{Net_i}) = \partial f(\mathrm{Net_i})/\partial \mathrm{Net_i}$

Backward Phase

- To update the weights of the **input-to-hidden** connections, we also follow gradient-descent method and the derivative chain rule

$$\Delta w_{qj} = -\eta \frac{\partial E}{\partial w_{qj}} = -\eta \left[\frac{\partial E}{\partial Out_q} \right] \left[\frac{\partial Out_q}{\partial Net_q} \right] \left[\frac{\partial Net_q}{\partial w_{qj}} \right]$$

- From the equation of the error function *E(**w**)*, it is clear that each error term $(d_i - y_i)$ $(i=1..n)$ is a function of Out_q

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{n} \left[d_i - f\left(\sum_{q=1}^{l} w_{iq} Out_q \right) \right]^2$$

Backward Phase

- Evaluating the derivative chain rule, we have

$$\Delta w_{qj} = \eta \sum_{i=1}^{n} \left[\left(d_i - Out_i \right) f'\left(Net_i \right) w_{iq} \right] f'\left(Net_q \right) x_j$$

$$= \eta \sum_{i=1}^{n} \left[\delta_i w_{iq} \right] f'\left(Net_q \right) x_j = \eta \delta_q x_j$$

- $\boldsymbol{\delta_q}$ is the **error signal** of neuron z_q in the **hidden layer**

$$\delta_q = -\frac{\partial E}{\partial Net_q} = -\left[\frac{\partial E}{\partial Out_q} \right] \left[\frac{\partial Out_q}{\partial Net_q} \right] = f'\left(Net_q \right) \sum_{i=1}^{n} \delta_i w_{iq}$$

where Net_q is the net input to neuron z_q in the hidden layer, and $f'(\mathrm{Net}_q)=\partial f(\mathrm{Net}_q)/\partial \mathrm{Net}_q$

Backward Phase

- According to the error equations δ_i and $\boldsymbol{\delta_q}$ above, the **error signal** of a neuron in a **hidden** layer is different from the error signal of a neuron in the **output** layer
- Because of this difference, the derived weight update procedure is called the *generalized delta learning rule*
- The **error signal** $\boldsymbol{\delta_q}$ of a **hidden** neuron z_q can be determined
  - in terms of the **error signals** δ_i of the neurons y_i (i.e., that z_q connects to) in the **output** layer
  - with the coefficients are just the weights w_{iq}
- The important feature of the BP algorithm: **the weights update rule is local**
  - To compute the weight change for a given connection, we need only the quantities available at both ends of that connection!

Backward Phase

- The discussed derivation can be easily extended to the network with more than one hidden layer by using the chain rule continuously
- The general form of the BP update rule is

$$\Delta w_{ab} = \eta \delta_a x_b$$

- *b* and *a* refer to the two ends of the (*b*→*a*) connection (i.e., from neuron (or input signal) *b* to neuron *a*)
- x_b is the output of the hidden neuron (or the input signal) *b*,
- δ_a is the error signal of neuron *a*

Back_propagation_incremental(D, η)

A network with Q feed-forward layers, $q = 1,2,...,Q$

${}^{q}Net_i$ and ${}^{q}Out_i$ are the net input and output of the i^{th} neuron in the q^{th} layer

The network has m input signals and n output neurons

${}^{q}w_{ij}$ is the weight of the connection from the j^{th} neuron in the $(q-1)^{th}$ layer to the i^{th} neuron in the q^{th} layer

Step 0 (Initialization)

Choose $E_{threshold}$ (a tolerable error)

Initialize the weights to small random values

Set E=0

Step 1 (Training loop)

Apply the input vector of the k^{th} training instance to the input layer (q=1)

${}^{q}Out_i = {}^{1}Out_i = x_i^{(k)}, \forall I$

Step 2 (Forward propagation)

Propagate the signal forward through the network, until the network outputs (in the output layer) ${}^{Q}Out_i$ have all been obtained

$${}^{q}Out_i = f\left({}^{q}Net_i\right) = f\left(\sum_j {}^{q}w_{ij}\, {}^{q-1}Out_j\right)$$

Step 3 (Output error measure)

Compute the error and error signals ${}^{Q}\delta_i$ for every neuron in the output layer

$$E = E + \frac{1}{2}\sum_{i=1}^{n}(d_i^{(k)} - {}^{Q}Out_i)^2$$

$${}^{Q}\delta_i = (d_i^{(k)} - {}^{Q}Out_i)f'({}^{Q}Net_i)$$

Step 4 (Error back-propagation)

Propagate the error backward to update the weights and compute the error signals ${}^{q-1}\delta_i$ for the preceding layers

$$\Delta^{q}w_{ij} = \eta.({}^{q}\delta_i).({}^{q-1}Out_j); \qquad {}^{q}w_{ij} = {}^{q}w_{ij} + \Delta^{q}w_{ij}$$

$${}^{q-1}\delta_i = f'({}^{q-1}Net_i)\sum_j {}^{q}w_{ji}\,{}^{q}\delta_j; \quad \text{for all } q = Q, Q-1, \ldots, 2$$

Step 5 (One epoch check)

Check whether the entire training set has been exploited (i.e., one epoch)

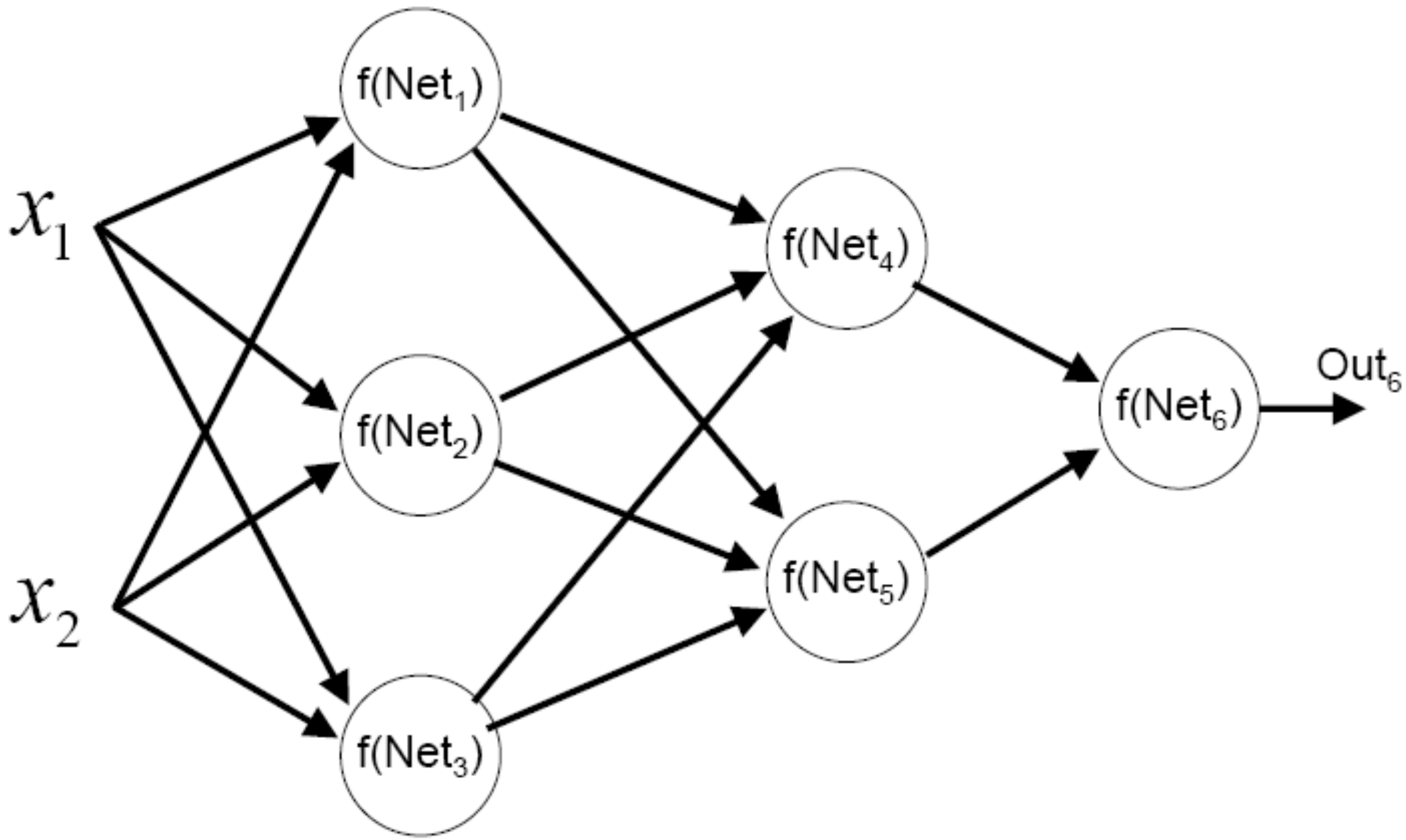
If the entire training set has been exploited, then go to step 6; otherwise, go to step 1

Step 6 (Total error check)

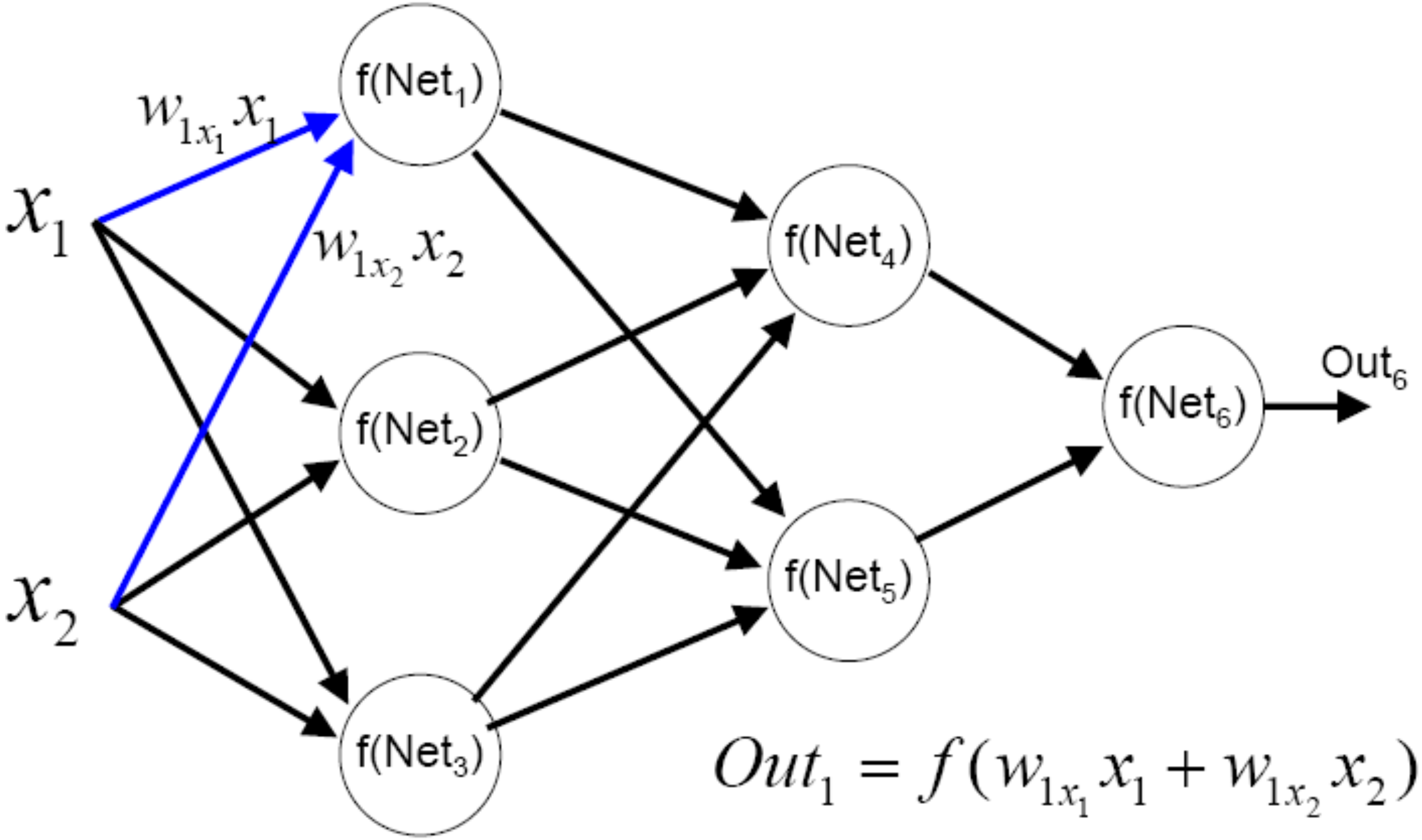
If the current total error is acceptable ($E<E_{threshold}$) then the training process terminates and output the final weights;

Otherwise, reset E=0, and initiate the new training epoch by going to step 1

Back Propagation algo – Ilustration

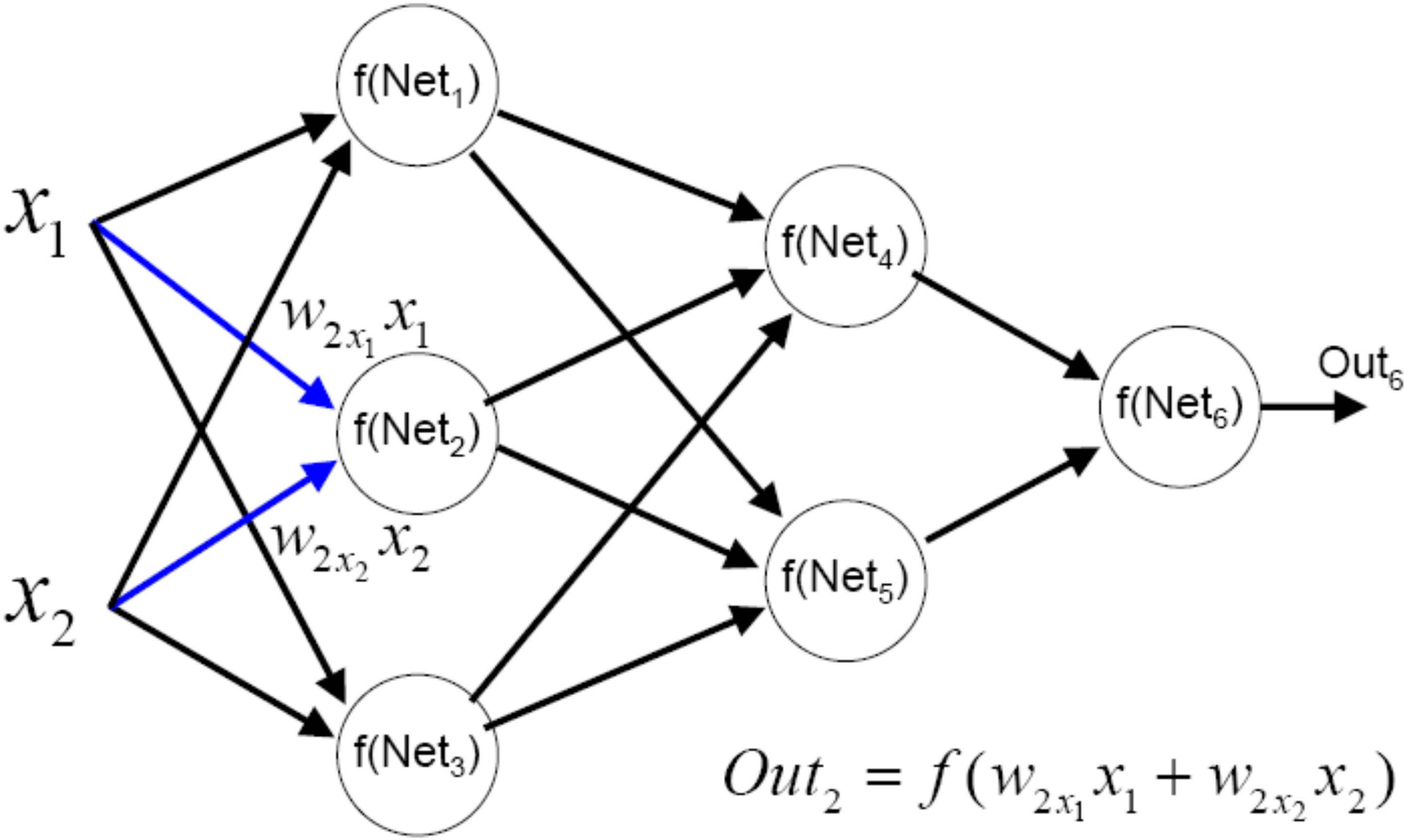


Forward phase



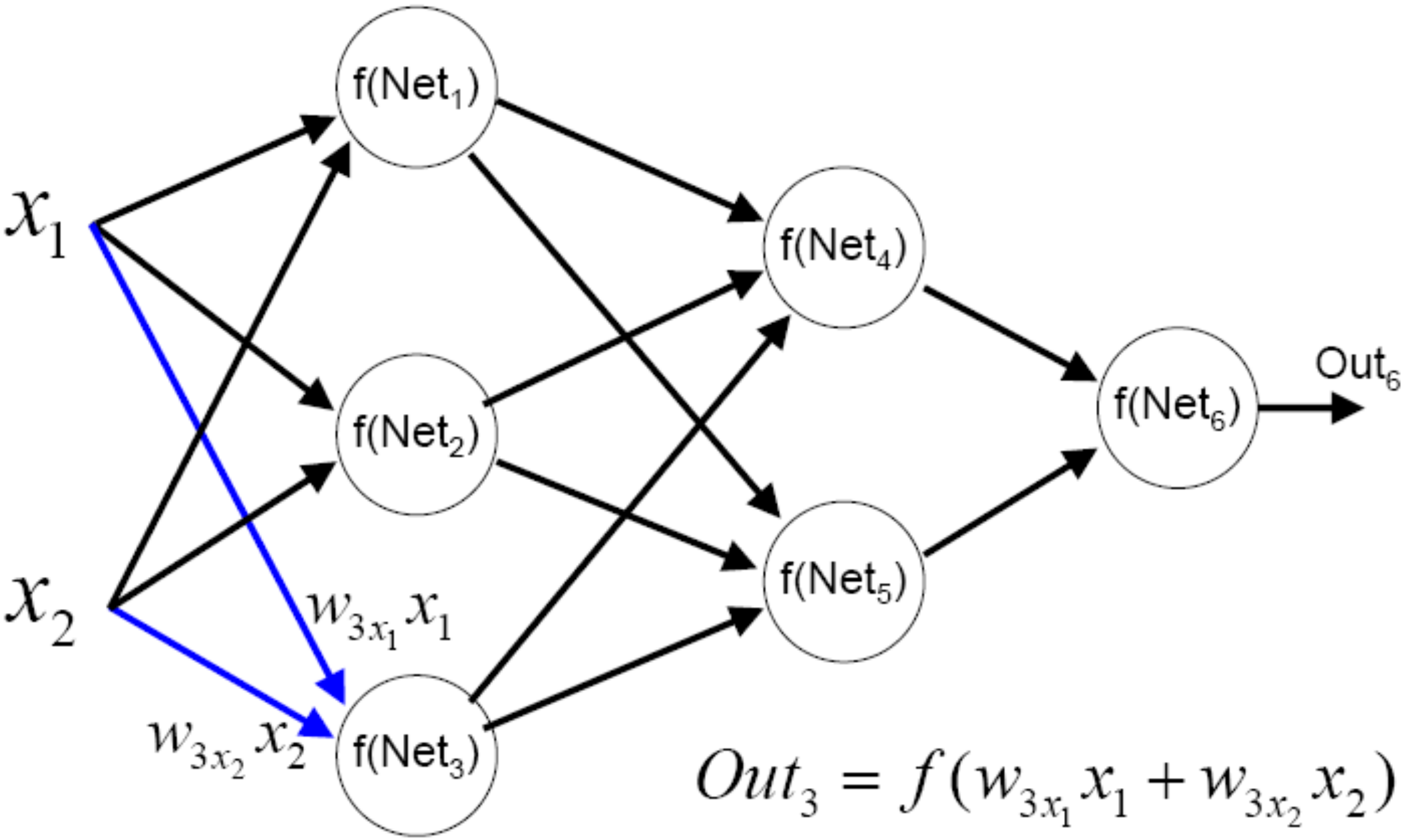
$$Out_1 = f(w_{1x_1}x_1 + w_{1x_2}x_2)$$

Forward phase



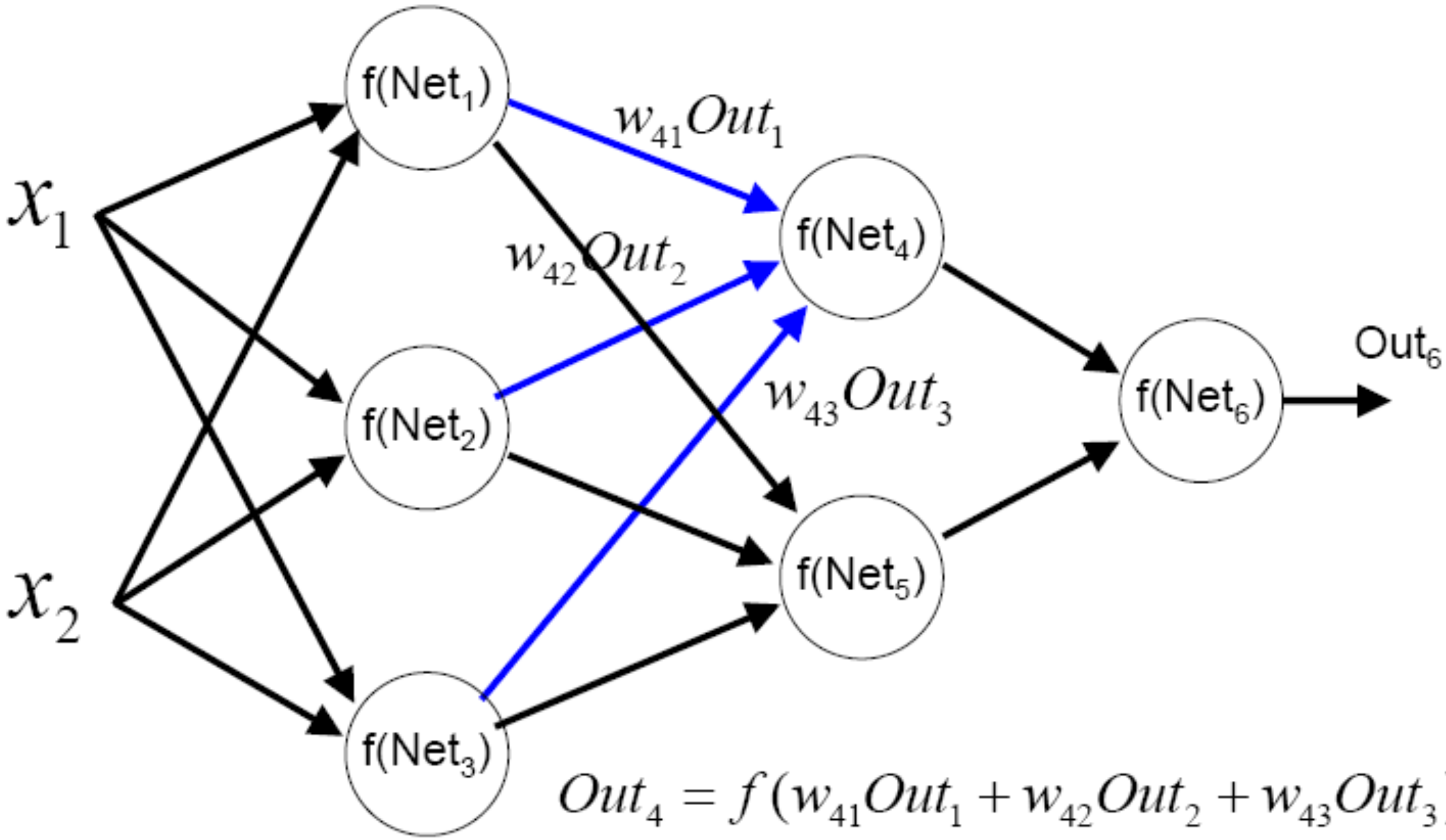
$$Out_2 = f(w_{2x_1} x_1 + w_{2x_2} x_2)$$

Forward phase



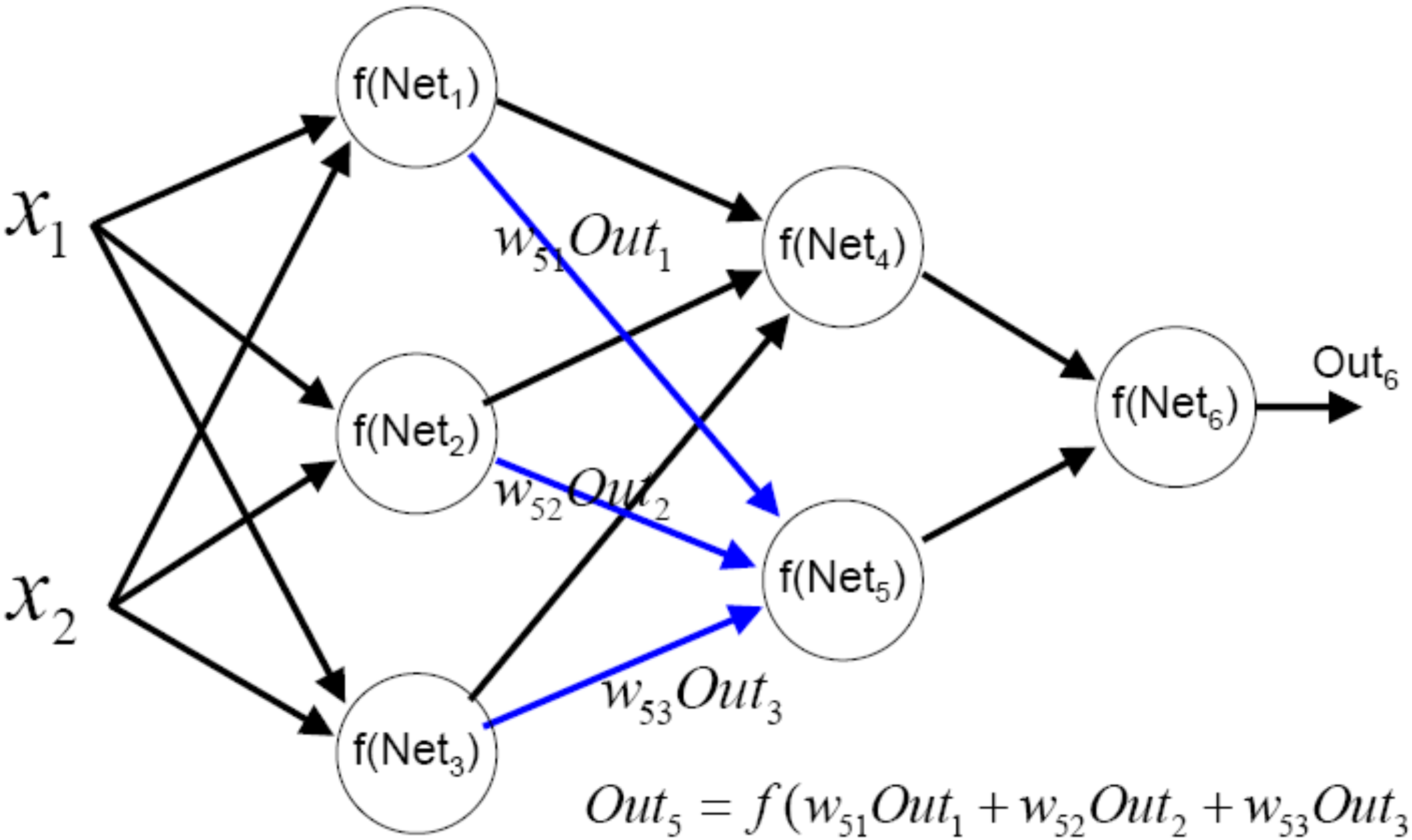
$$Out_3 = f(w_{3x_1}x_1 + w_{3x_2}x_2)$$

Forward phase



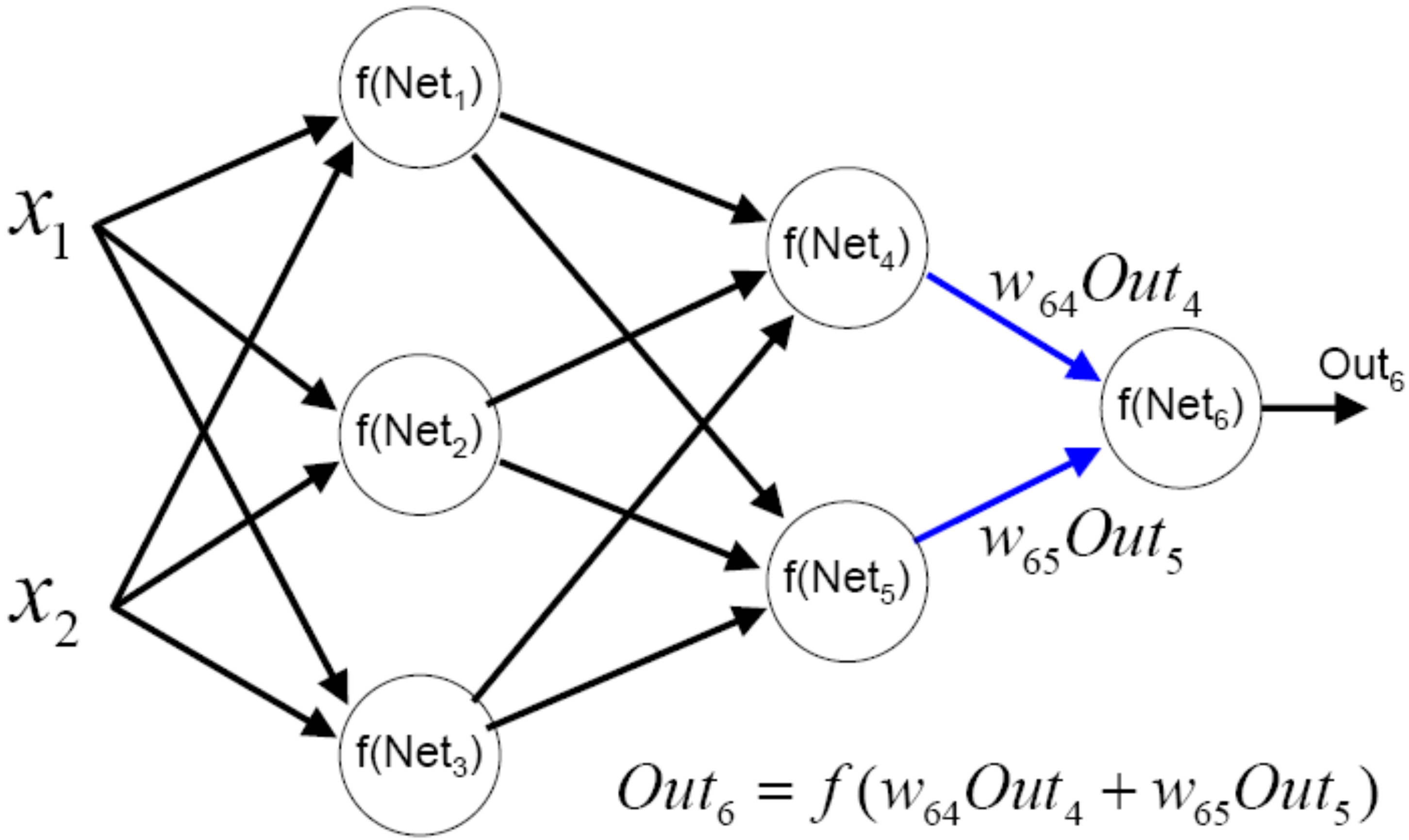
$$Out_4 = f(w_{41}Out_1 + w_{42}Out_2 + w_{43}Out_3)$$

Forward phase



$$Out_5 = f(w_{51}Out_1 + w_{52}Out_2 + w_{53}Out_3)$$

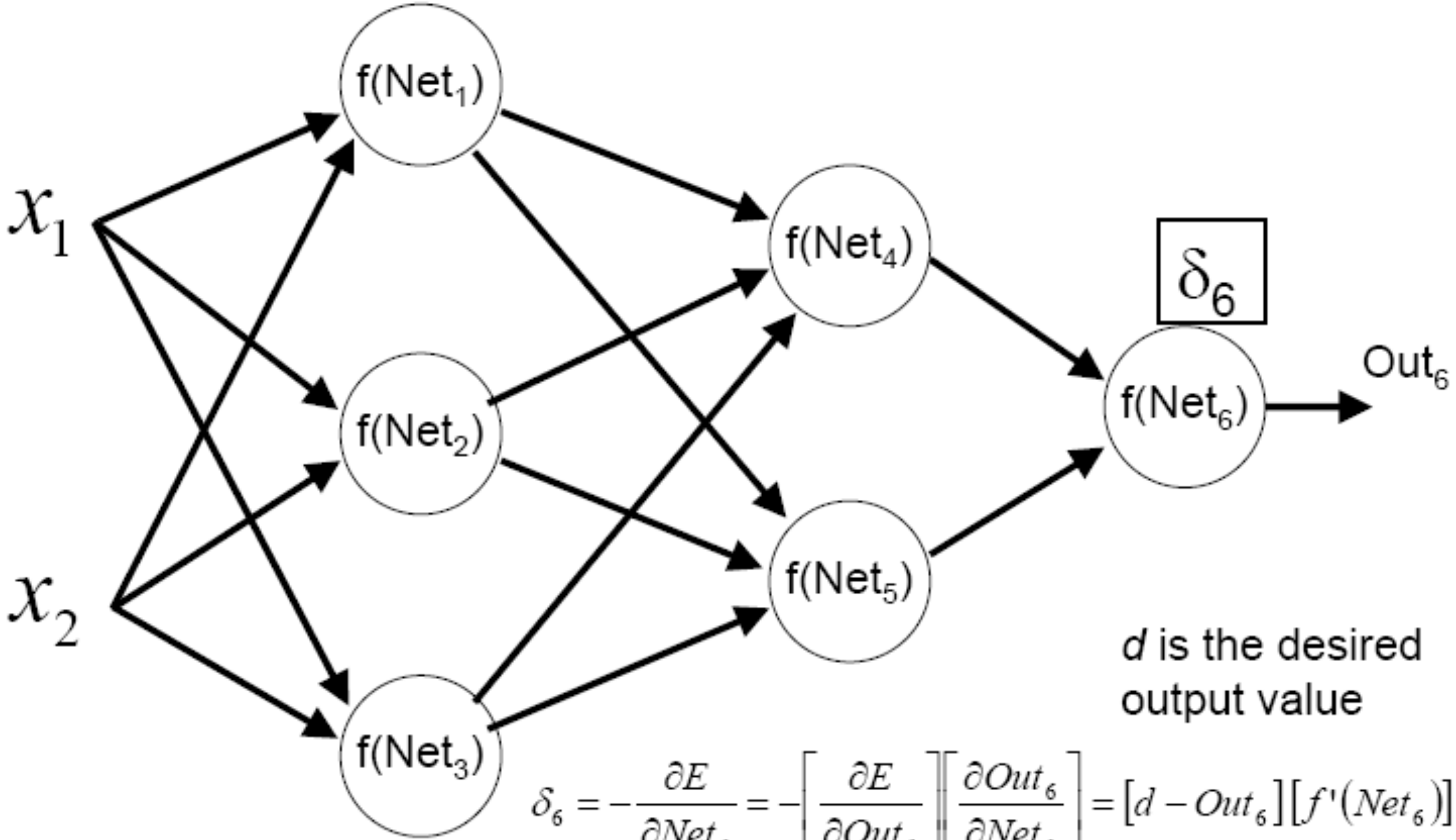
Forward phase



$$x_1 \quad x_2 \quad w_{64}Out_4 \quad w_{65}Out_5$$

$$Out_6 = f(w_{64}Out_4 + w_{65}Out_5)$$

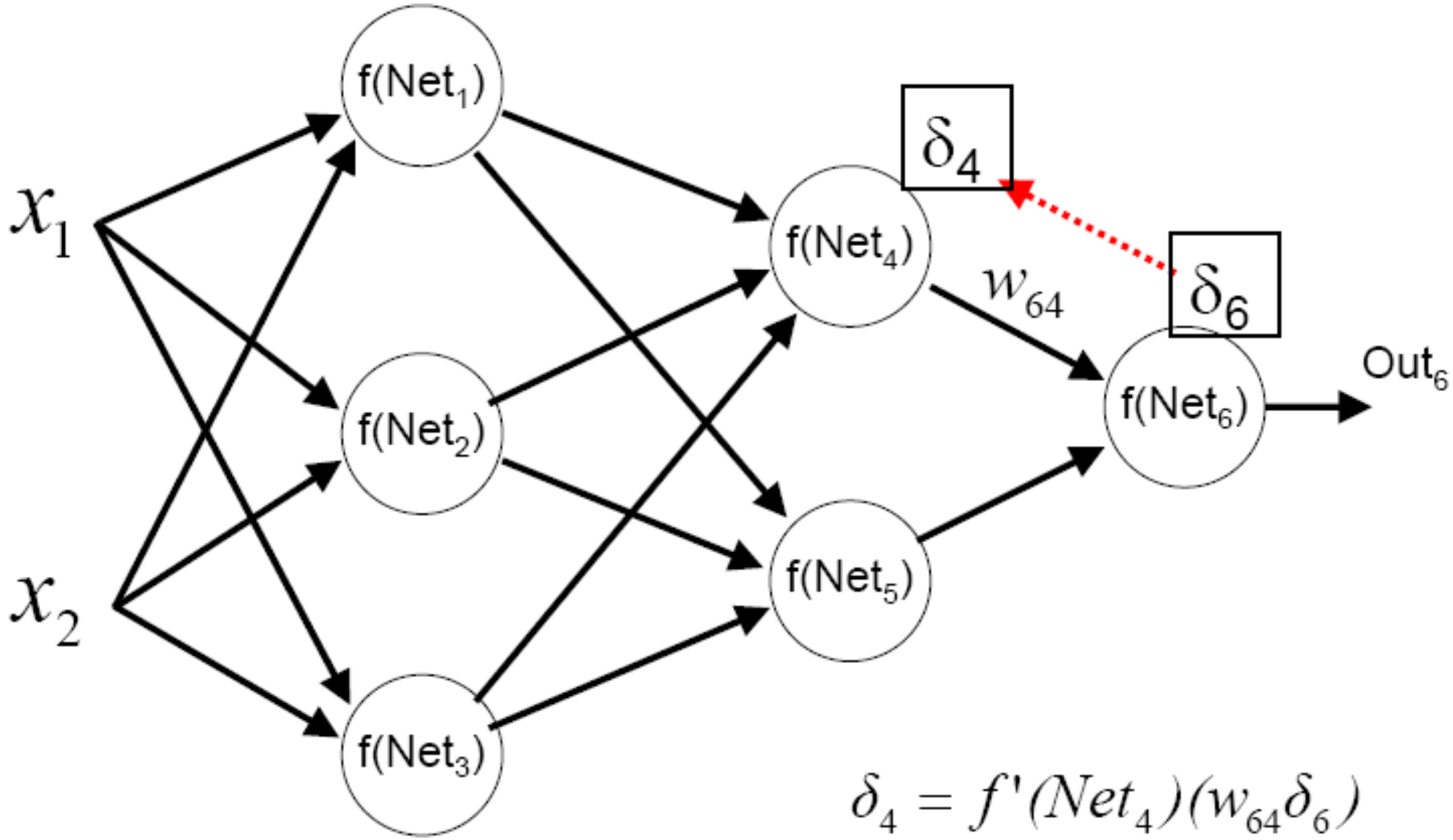
Computing error



d is the desired output value

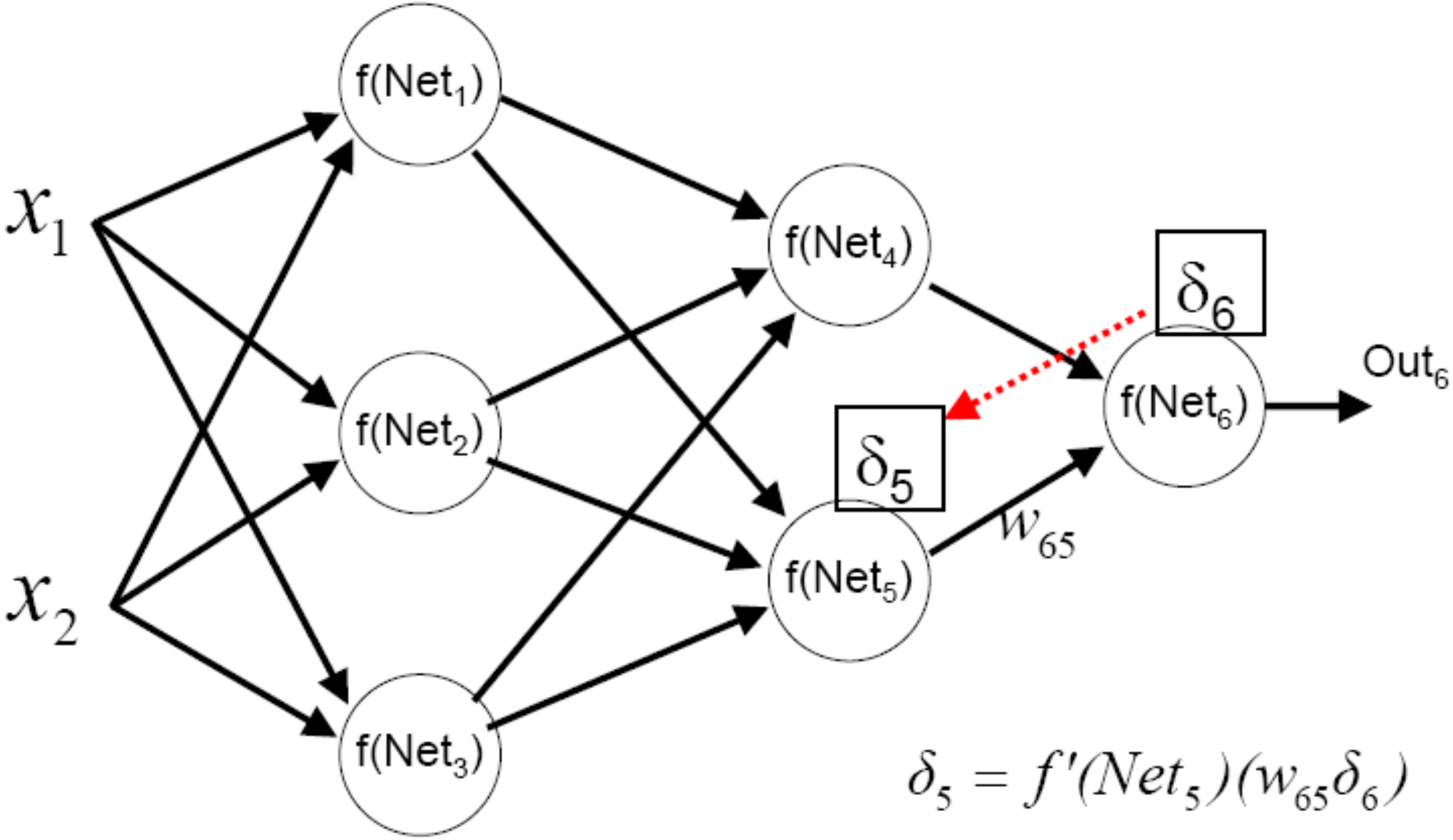
$$\delta_6 = -\frac{\partial E}{\partial Net_6} = -\left[\frac{\partial E}{\partial Out_6}\right]\left[\frac{\partial Out_6}{\partial Net_6}\right] = [d - Out_6][f'(Net_6)]$$

Backward phase



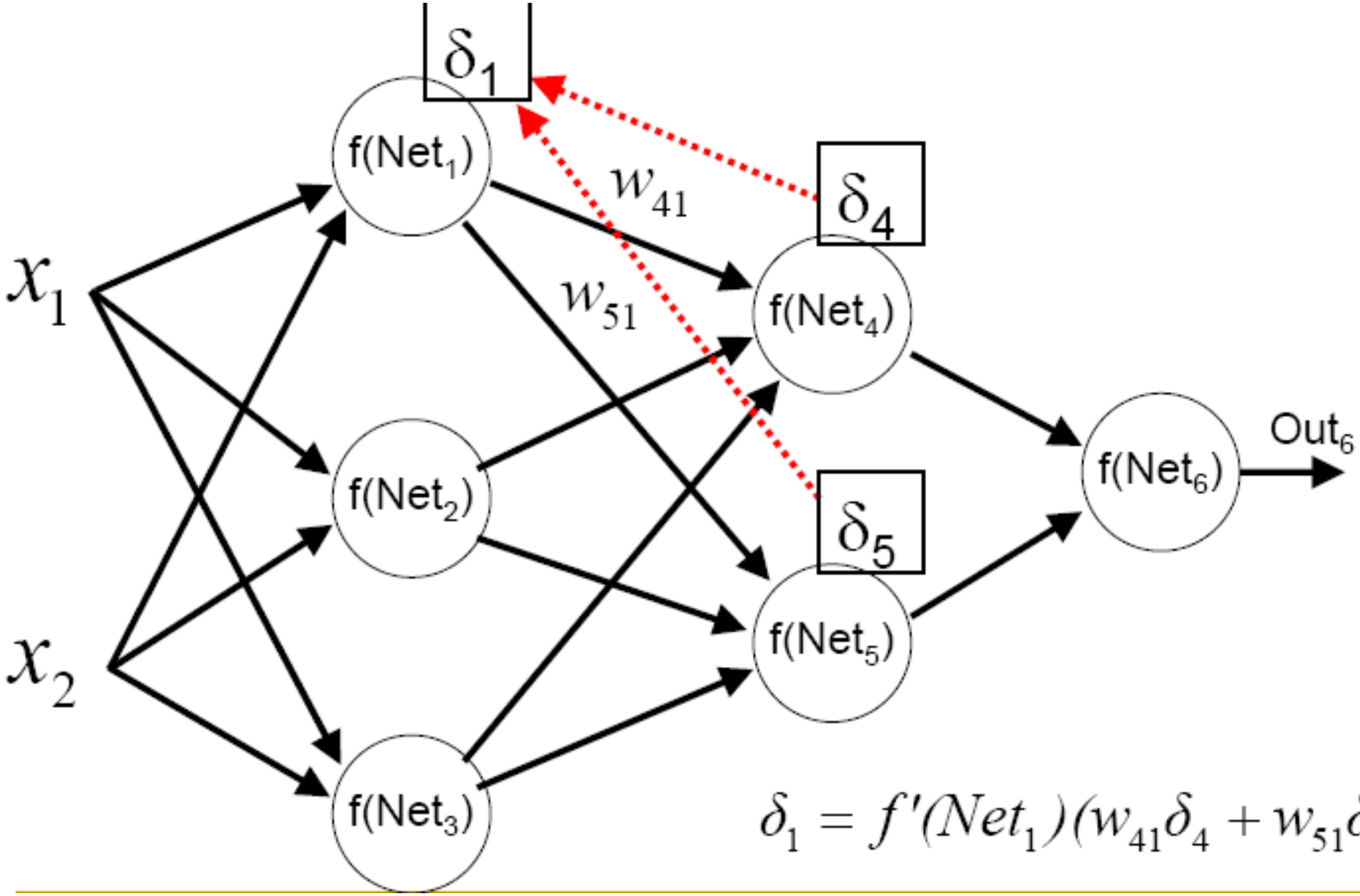
$$\delta_4 = f'(Net_4)(w_{64}\delta_6)$$

Backward phase



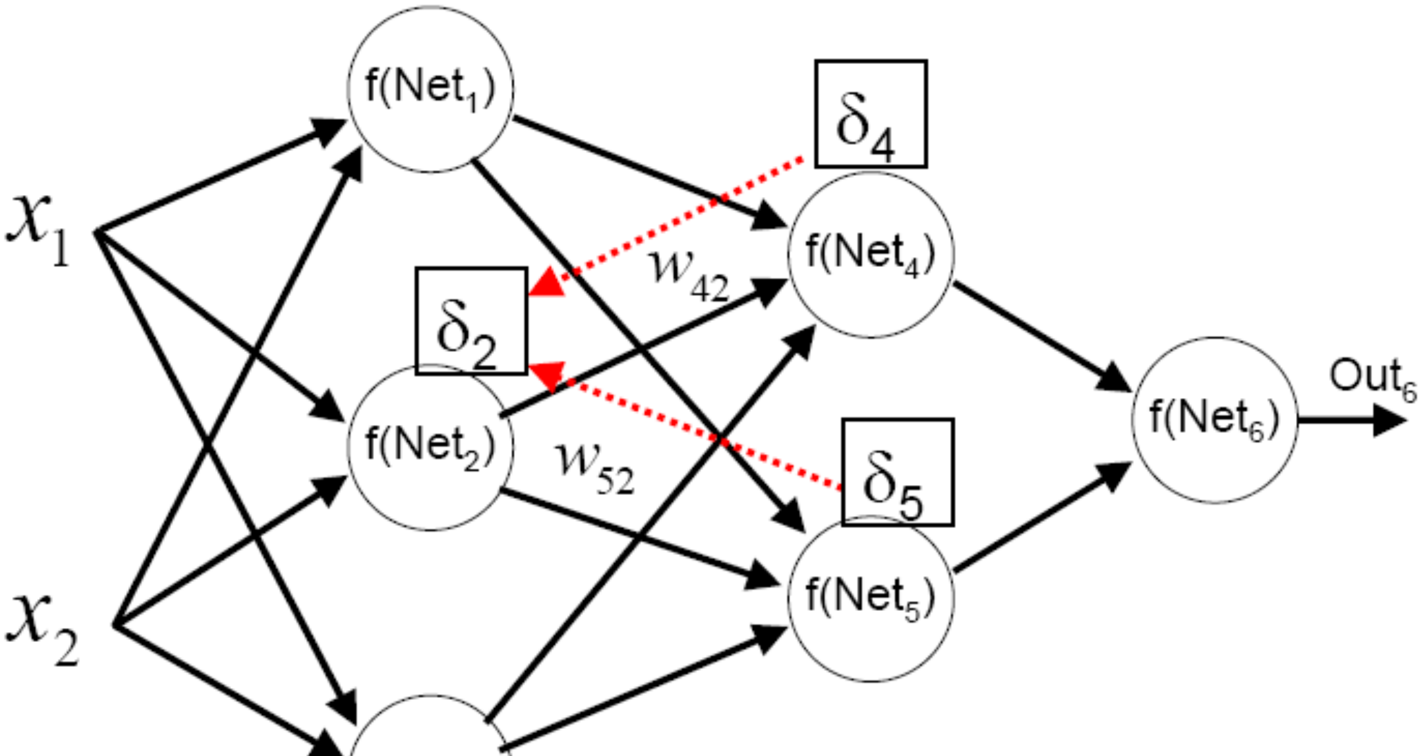
$$\delta_5 = f'(Net_5)(w_{65}\delta_6)$$

Backward phase



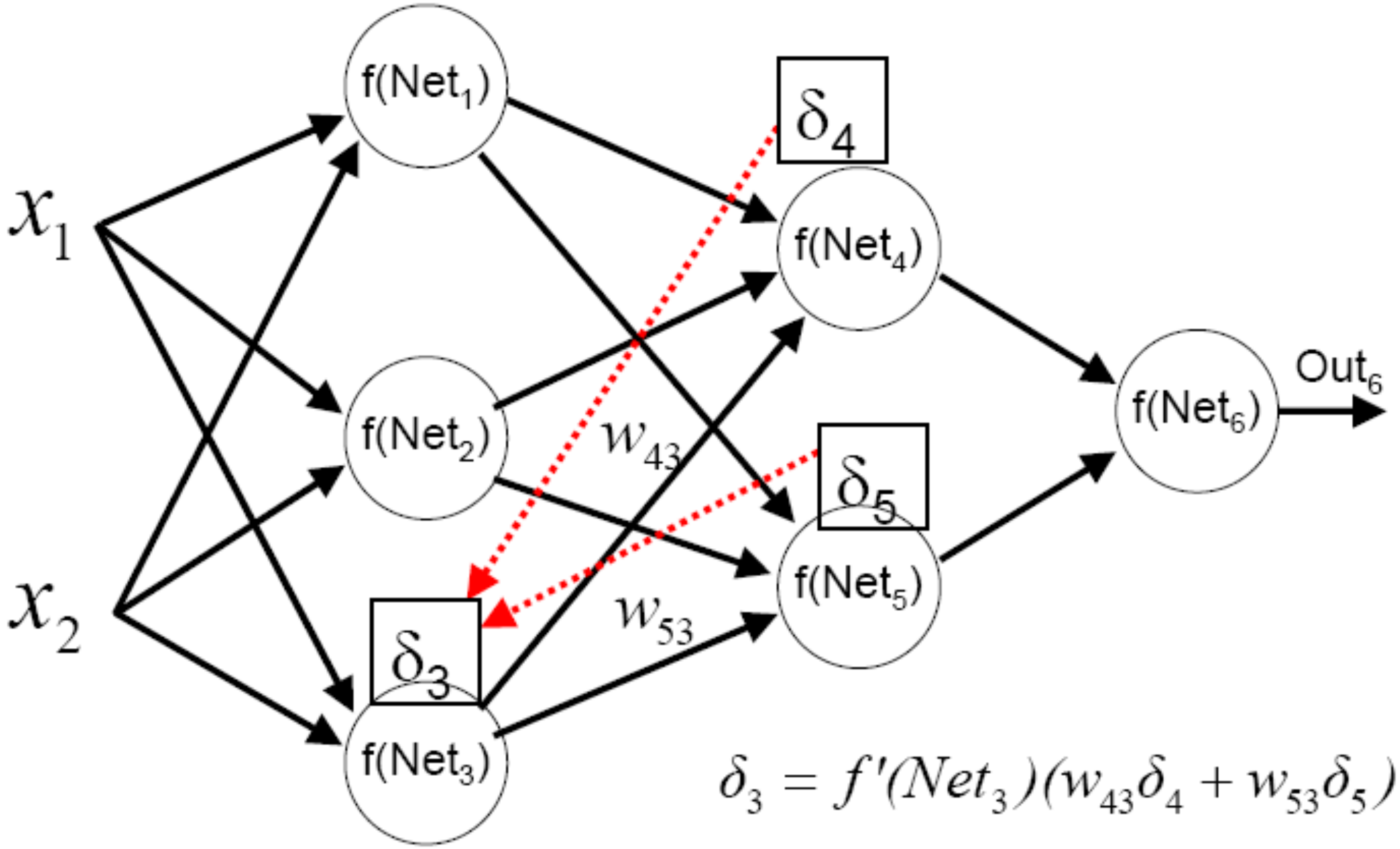
$$\delta_1 = f'(Net_1)(w_{41}\delta_4 + w_{51}\delta_5)$$

Backward phase



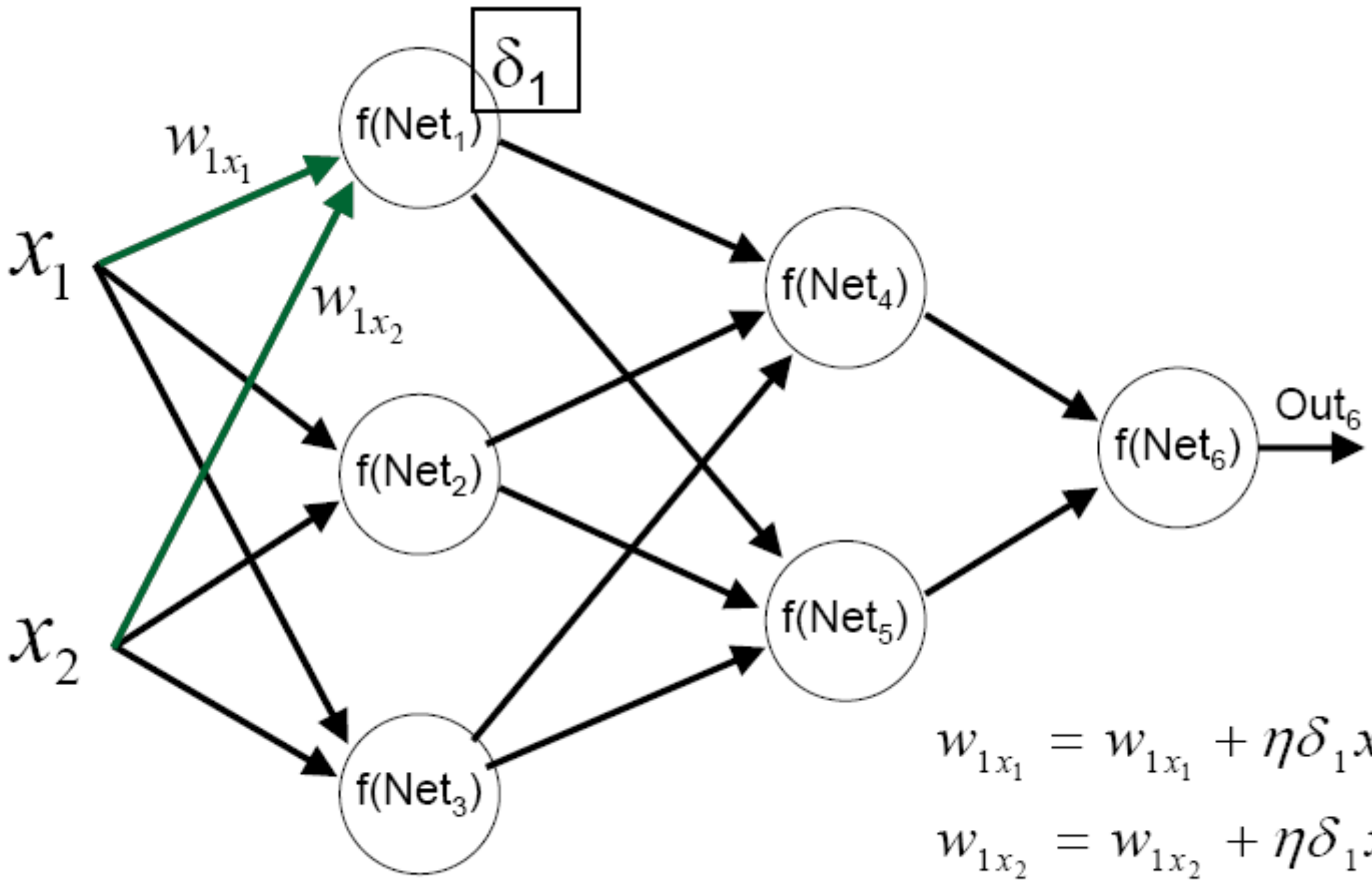
$$\delta_2 = f'(Net_2)(w_{42}\delta_4 + w_{52}\delta_5)$$

Backward phase



$$\delta_3 = f'(Net_3)(w_{43}\delta_4 + w_{53}\delta_5)$$

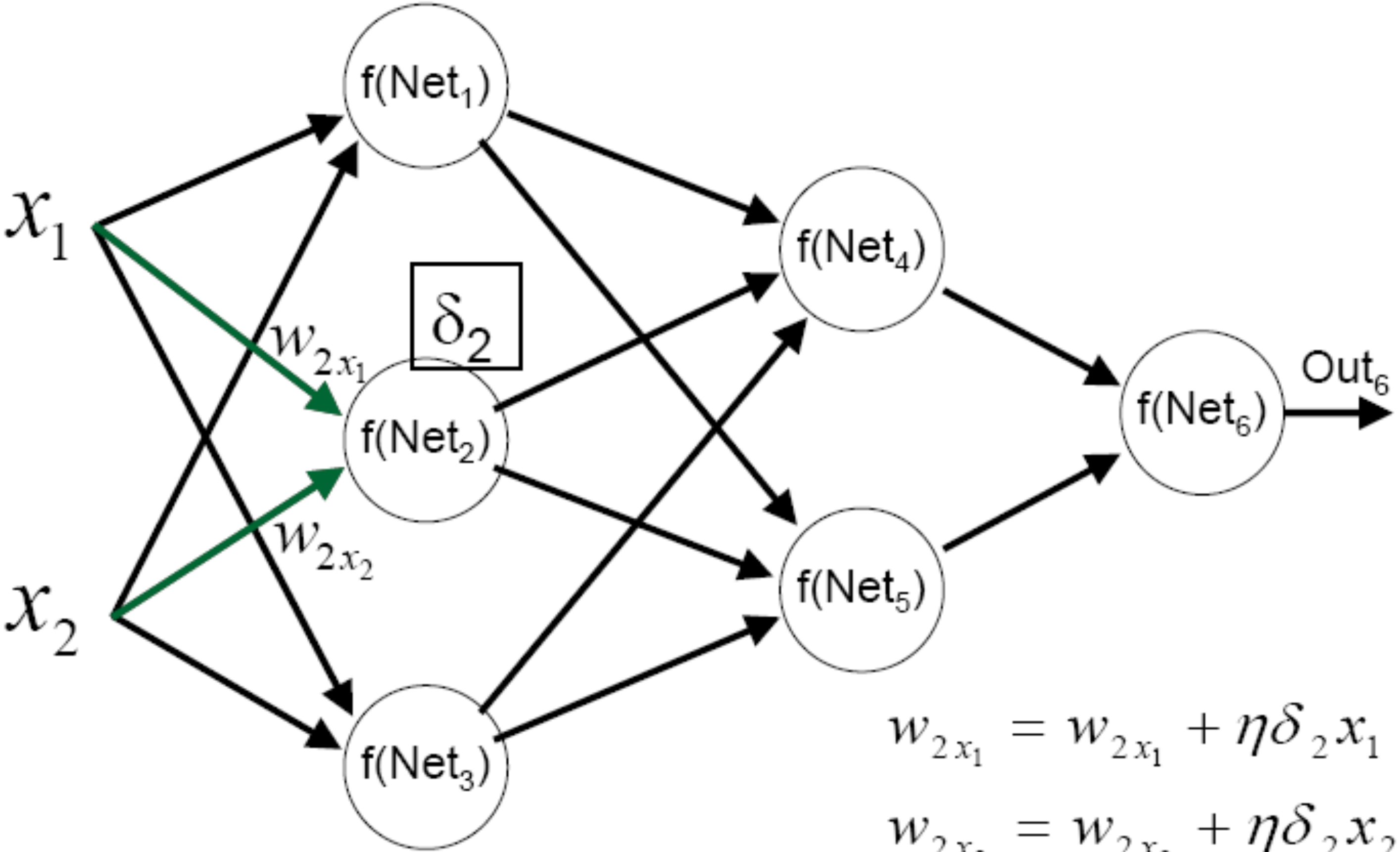
Weight Update



$$w_{1x_1} = w_{1x_1} + \eta\delta_1 x_1$$

$$w_{1x_2} = w_{1x_2} + \eta\delta_1 x_2$$

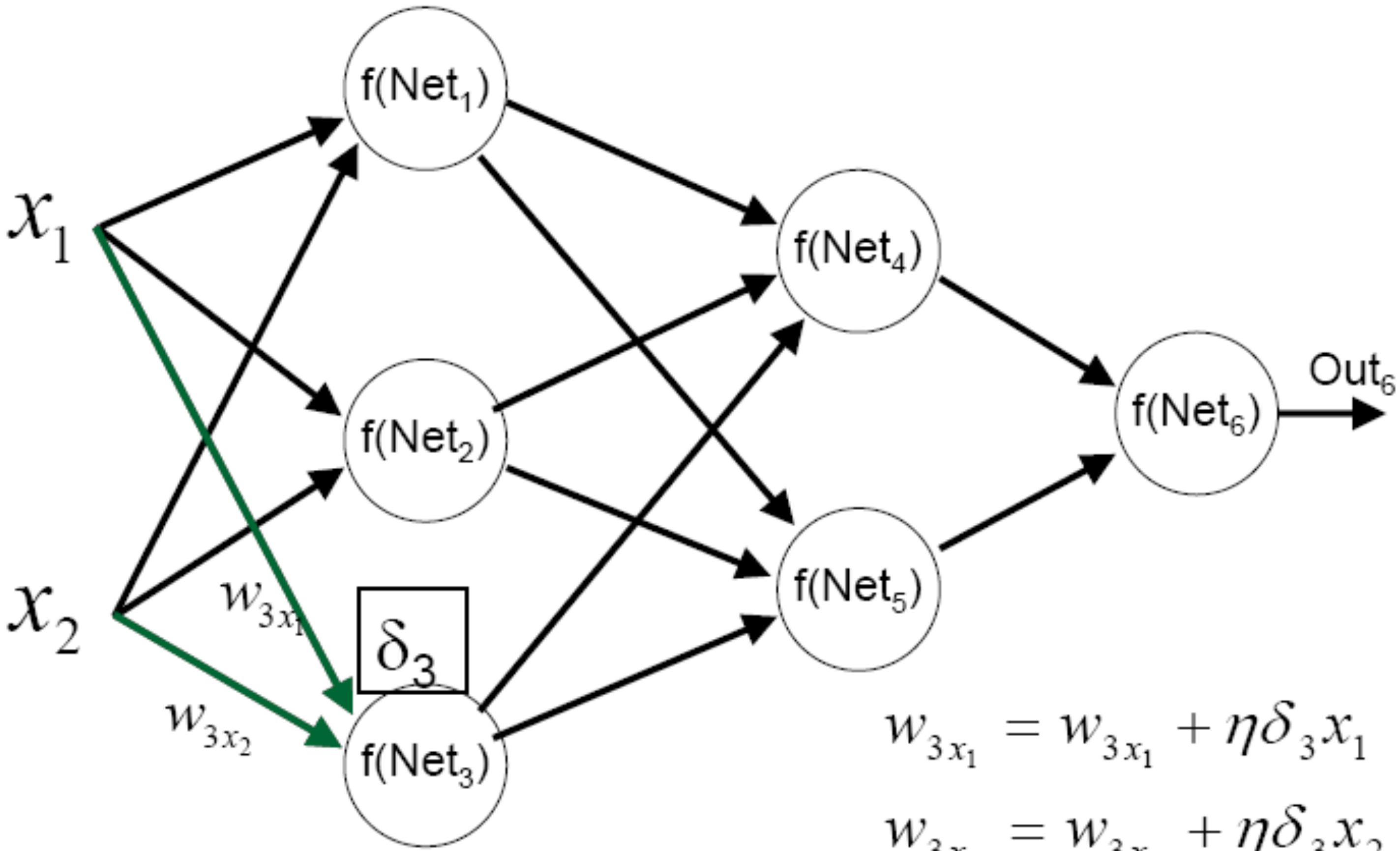
Weight Update



$$w_{2x_1} = w_{2x_1} + \eta\delta_2 x_1$$

$$w_{2x_2} = w_{2x_2} + \eta\delta_2 x_2$$

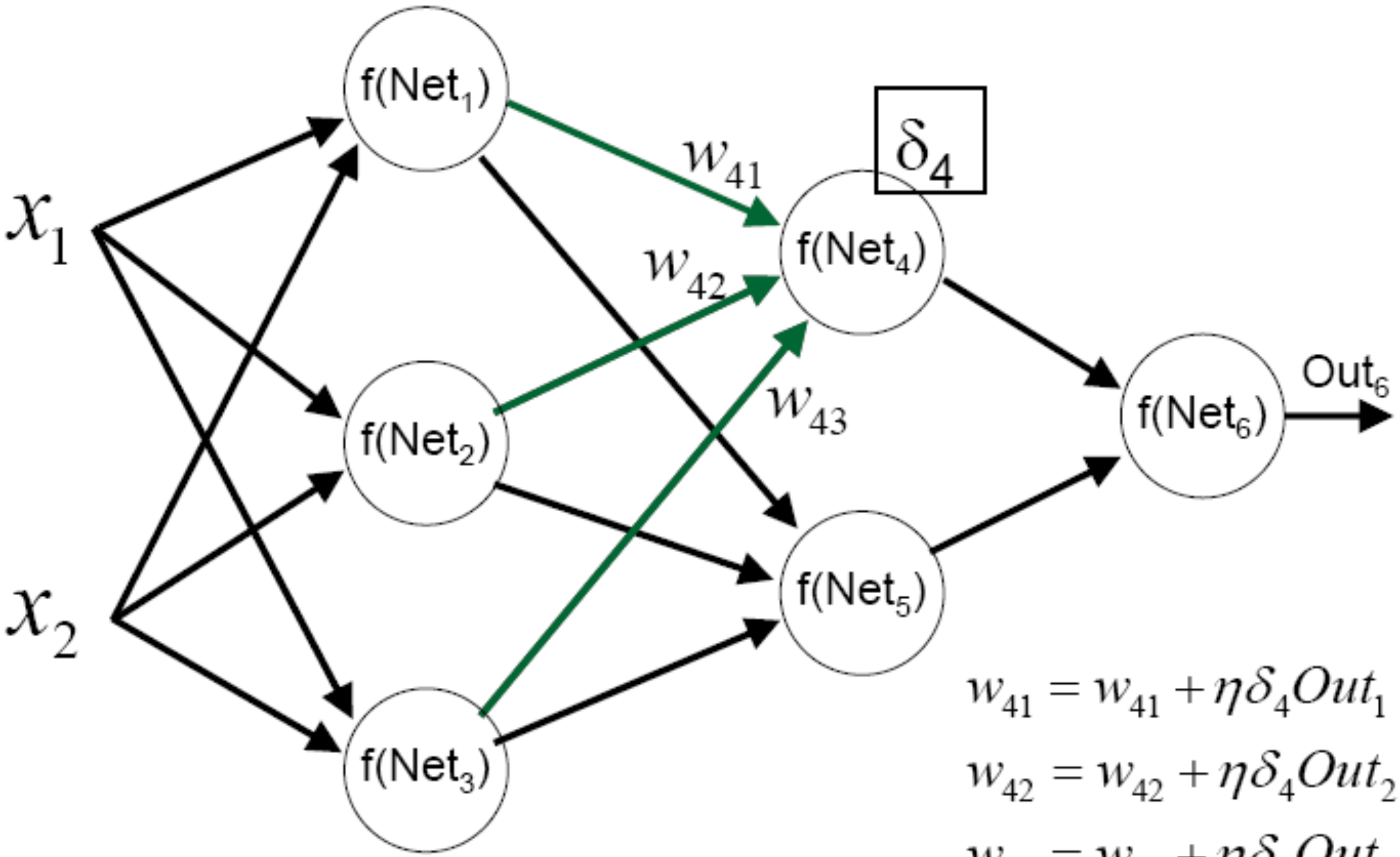
Weight Update



$$w_{3x_1} = w_{3x_1} + \eta\delta_3 x_1$$

$$w_{3x_2} = w_{3x_2} + \eta\delta_3 x_2$$

Weight Update

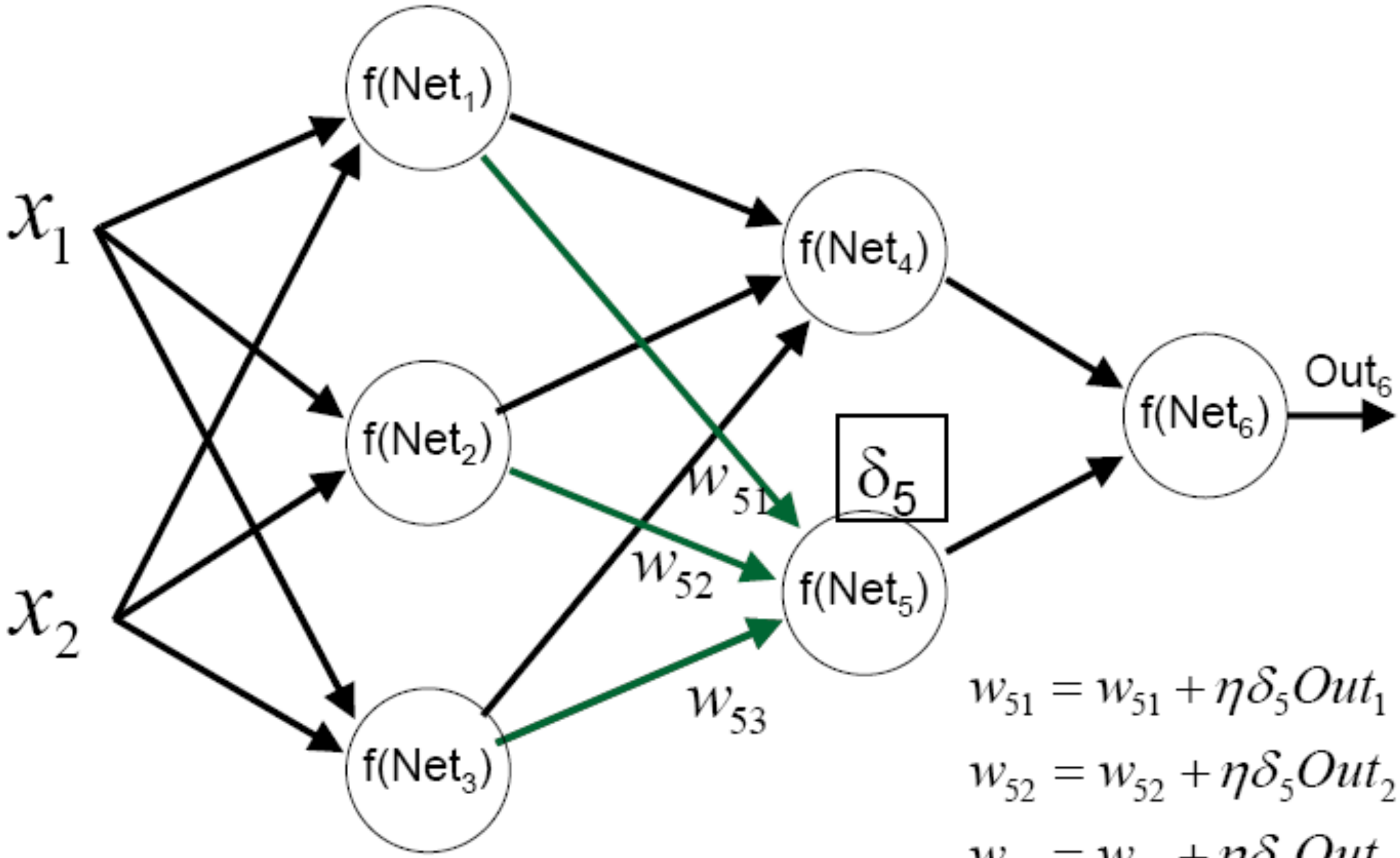


$$w_{41} = w_{41} + \eta\delta_4 Out_1$$

$$w_{42} = w_{42} + \eta\delta_4 Out_2$$

$$w_{43} = w_{43} + \eta\delta_4 Out_3$$

Weight Update

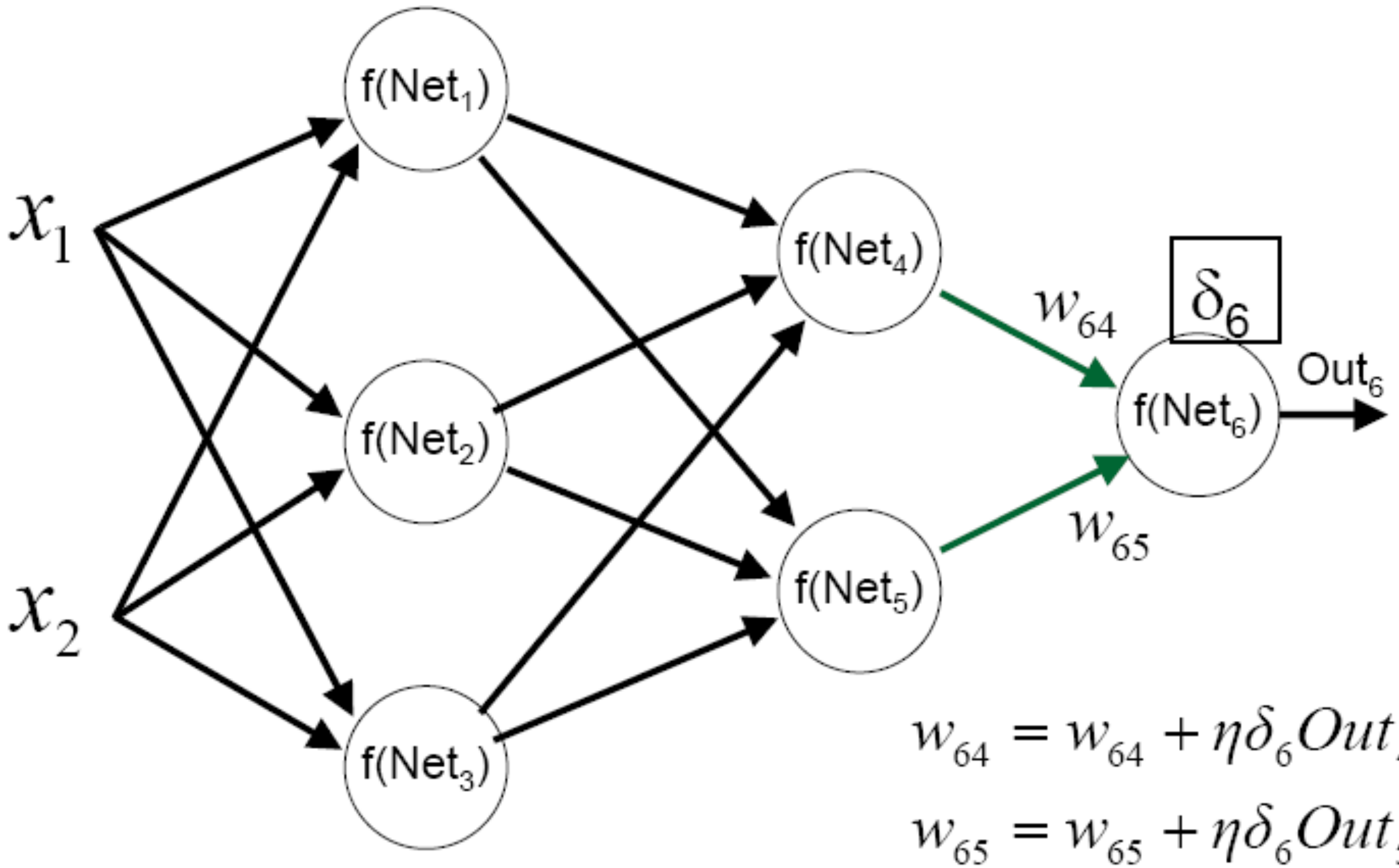


$$w_{51} = w_{51} + \eta\delta_5 Out_1$$

$$w_{52} = w_{52} + \eta\delta_5 Out_2$$

$$w_{53} = w_{53} + \eta\delta_5 Out_3$$

Weight Update



$$w_{64} = w_{64} + \eta \delta_6 Out_4$$

$$w_{65} = w_{65} + \eta \delta_6 Out_5$$

Issues- Initial weight

- Often the weights are initially set to small random values
- If the initial weights are large
  - the sigmoid functions will saturate from the beginning
  - the system gets stuck at a local minimum or in a very flat plateau near the starting point
- Some suggestions for w^0_{ab} (from neuron b to neuron a)
  - Let's denote n_a is the number of neurons in the layer of neuron a

$$w^0_{ab} \in [-1/n_a, 1/n_a]$$

  - Let's denote k_a the number of neurons that feed-forward to neuron a (the number of input links of neuron a)

$$w^0_{ab} \in [-3/\sqrt{k_a}, 3/\sqrt{k_a}]$$

Issues – Learning Rate

- Significantly affects the effectiveness and convergence of the BP learning algorithm
  - A large value of η could speed up the convergence, but might result in overshooting or local minima
  - a smaller value of η may take a very long time for the training
- Usually chosen experimentally for each problem
- Good values of the learning rate at the beginning of the training may not be as good in later time (of the training)
  - To use an adaptive (dynamic) learning rate
- After an update of the weights update, check whether the weight update results in a decrease of the error

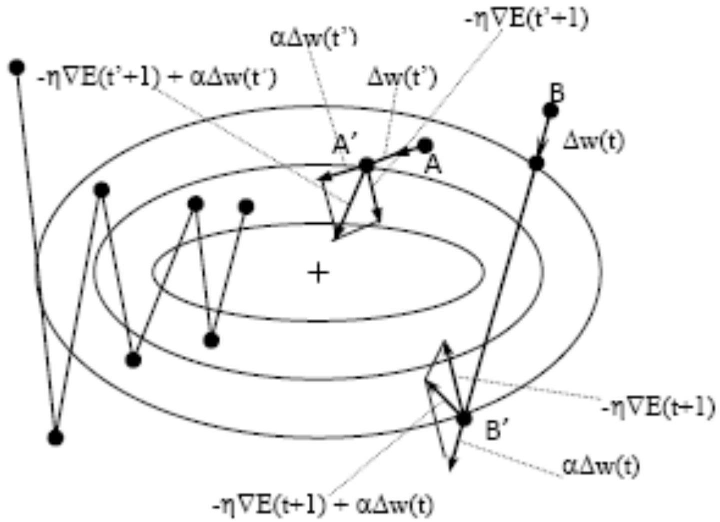
$$\Delta\eta = \begin{cases} a & \text{, if } \Delta E < 0 \text{ consistently} \\ -b\eta & \text{, if } \Delta E > 0 \\ 0 & \text{, otherwise.} \end{cases} \qquad (a, b > 0)$$

Issues- Momentum

- *Gradient descent* can be very slow if the learning rate η is small and can oscillate widely if η is too large
- To reduce the oscillation, incorporate a momentum term in the normal gradient-descent

$$\Delta w^{(t)} = -\eta \nabla E^{(t)} + \alpha \Delta w^{(t-1)}$$

  where α ($\in$[0,1]) is a momentum parameter, and a value of 0.9 is often used
- One rule, based on the experiments, to choose the properly learning rate and momentum is: $(\eta+\alpha) >\approx 1$; where $\alpha>\eta$ to prevent the oscillation



Gradient descent on a simple quadratic surface. There is no momentum on the left trajectory, while on the right there is a momentum term.

Issues – Number of hidden neurons

- The size of a hidden layer is a fundamental question for the application of multilayer feed-forward NNs to real-world problems
- It practice, it is very difficult to determine a sufficient number of neurons to achieve the desired accuracy
- The size of a hidden layer is usually determined experimentally – trial and test
- A recommendation
  - Begin with the size of hidden nodes ~ a relatively small fraction of the dimensionality of the input layer
  - If the network fails to converge to a solution, add more hidden nodes
  - If it does converge, you may try fewer hidden nodes

Advantages & Disadvantages

- Advantages
  - Massively parallel in nature
  - Fault (noise) tolerant because of parallelism
  - Can be designed to be adaptive
- Disadvantages
  - No clear rules or design guidelines for arbitrary applications
  - No general way to assess the internal operation of the network
  - (therefore, an ANN system is seen as a "black-box")
  - Difficult to predict future network performance (generalization)

ANN- When ?

- Input is high-dimensional discrete or real-valued
- The target function is real-valued, discrete-valued or vector-valued
- Possibly noisy data
- The form of the target function is unknown
- Human readability of result is not (very) important
- Long training time is accepted
- Short classification/prediction time is required