ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Artificial Intelligence

Lecturer 15 – Artificial Neuron Networks

School of Information and Communication Technology - HUST

Artificial neural networks

- Artificial neural network (ANN)
  - Inspired by biological neural systems, i.e., human brains
  - ANN is a network composed of a number of artificial neurons
- Neuron
  - Has an input/output (I/O) characteristic
  - Implements a local computation
- The output of a unit is determined by
  - Its I/O characteristic
  - Its interconnections to other units
  - Possibly external inputs

Artificial neural networks

- ANN can be seen as a parallel distributed information processing structure
- ANN has the ability to learn, recall, and generalize from training data by assigning and adjusting the interconnection weights
- The overall function is determined by
  - The network topology
  - The individual neuron characteristic
  - The learning/training strategy
  - The training data

Applications of ANNs

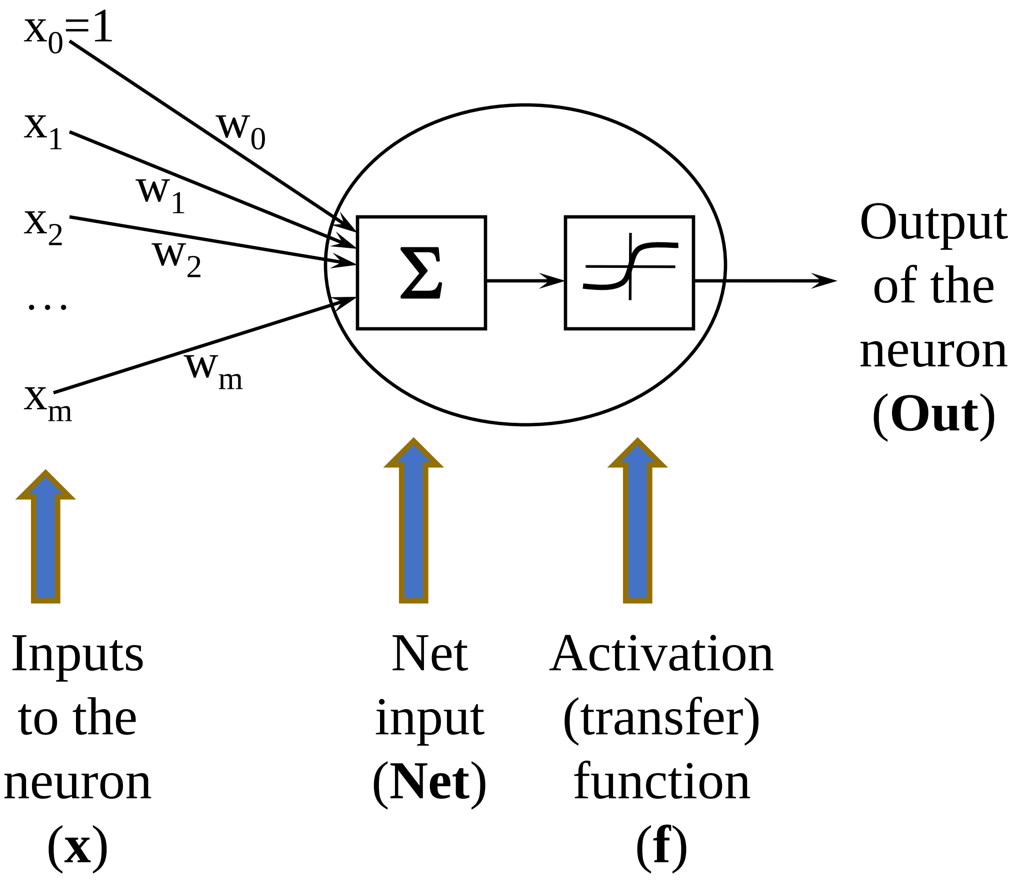
- Image processing and computer vision
  - E.g., image matching, preprocessing, segmentation and analysis, computer vision, image compression, stereo vision, and processing and understanding of time-varying images
- Signal processing
  - E.g., seismic signal analysis and morphology
- Pattern recognition
  - E.g., feature extraction, radar signal classification and analysis, speech recognition and understanding, fingerprint identification, character recognition, face recognition, and handwriting analysis
- Medicine
  - E.g., electrocardiographic signal analysis and understanding, diagnosis of various diseases, and medical image processing

Applications of ANNs

- Military systems
  - E.g., undersea mine detection, radar clutter classification, and tactical speaker recognition
- Financial systems
  - E.g., stock market analysis, real estate appraisal, credit card authorization, and securities trading
- Planning, control, and search
  - E.g., parallel implementation of constraint satisfaction problems, solutions to Traveling Salesman, and control and robotics
- Power systems
  - E.g., system state estimation, transient detection and classification, fault detection and recovery, load forecasting, and security assessment
- ...

Structure and operation of a neuron

- The **input signals** to the neuron (x_i, i = 1..m)
  - Each input x_i associates with a weight w_i
- The **bias** w_0 (with the input x_0=1)
- **Net input** is an integration function of the inputs – `Net(`**`w`**`,`**`x`**`)`
- **Activation (transfer) function** computes the output of the neuron – `f(Net(`**`w`**`,`**`x`**`))`
- **Output** of the neuron: `Out=f(Net(`**`w`**`,`**`x`**`))`

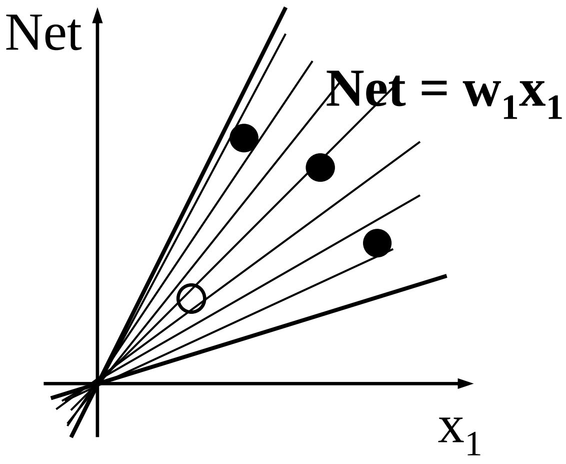


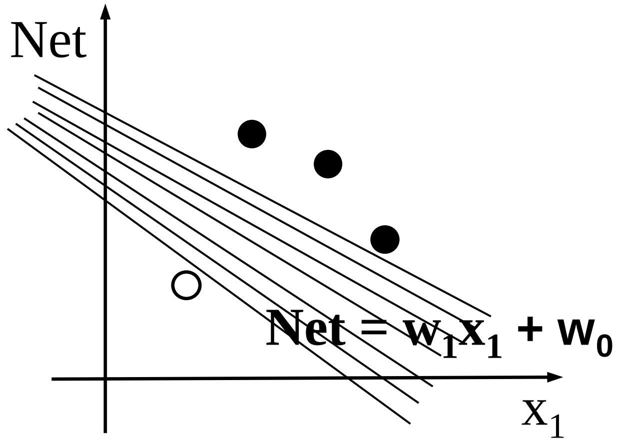
Net input and The bias

- The net input is typically computed using a linear function

$$Net = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_m x_m = w_0.1 + \sum_{i=1}^{m} w_i x_i = \sum_{i=0}^{m} w_i x_i$$

- The importance of the bias (w_0)
  - → The family of separation functions $Net=w_1x_1$ cannot separate the instances into two classes
  - → The family of functions $Net=w_1x_1+w_0$ can





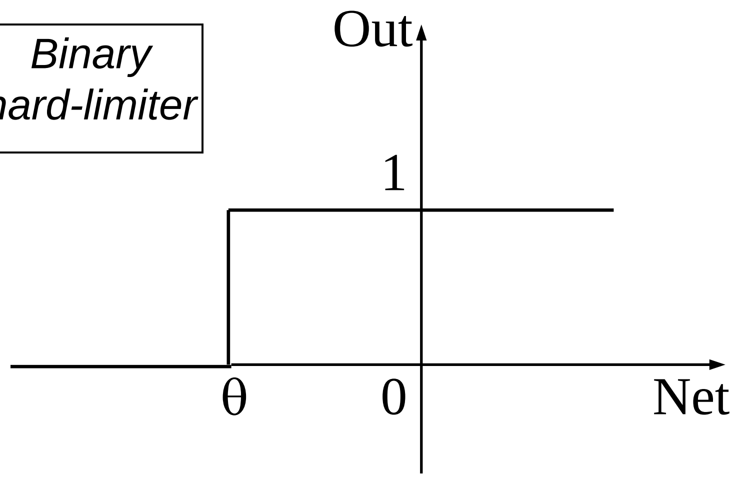
Activation function – Hard-limiter

- Also called the threshold function
- The output of the hard-limiter is either of the two values
- θ is the threshold value
- **Disadvantage**: neither continuous nor continuously differentiable

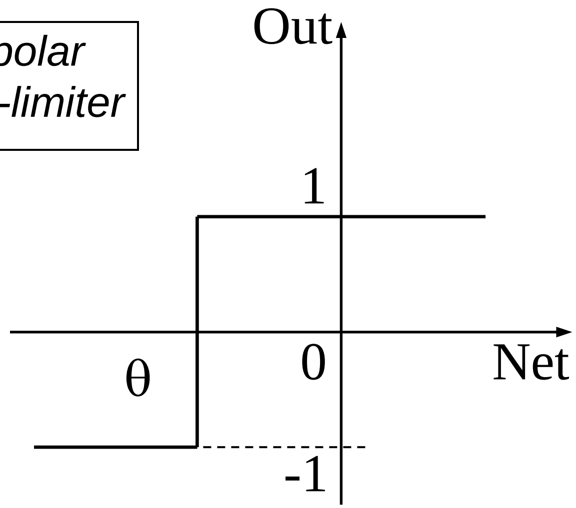
$$Out(Net) = hl1(Net, \theta) = \begin{cases} 1, \text{ if} & Net \geq \theta \\ 0, \text{ if} & \text{otherwise} \end{cases}$$

$$Out(Net) = hl2(Net, \theta) = sign(Net, \theta)$$

Binary hard-limiter



Bipolar hard-limiter



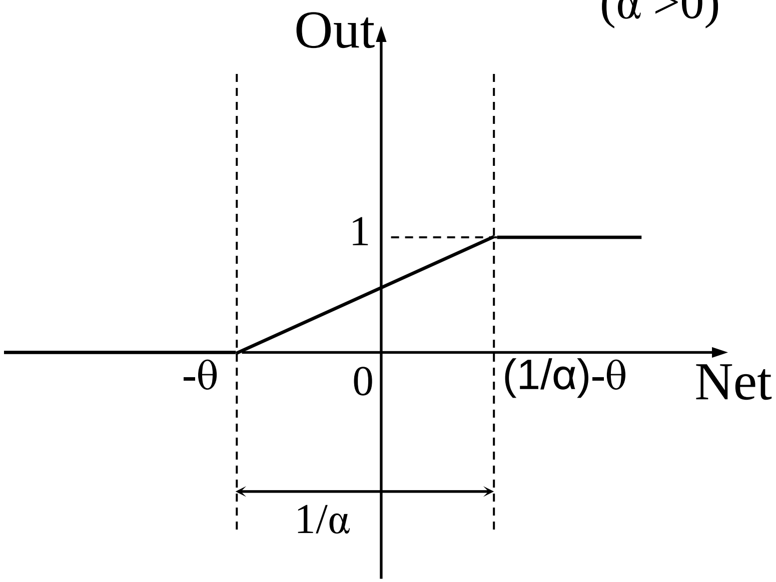
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Activation function – Threshold logic

$$Out(Net) = tl(Net, \alpha, \theta) = \begin{cases} 0, \text{ if} & Net < -\theta \\ \alpha(Net + \theta), \text{if} & -\theta \le Net \le \dfrac{1}{\alpha} - \theta \\ 1, \text{ if} & Net > \dfrac{1}{\alpha} - \theta \end{cases}$$

$$= \max(0, \min(1, \alpha(Net + \theta)))$$

(α >0)

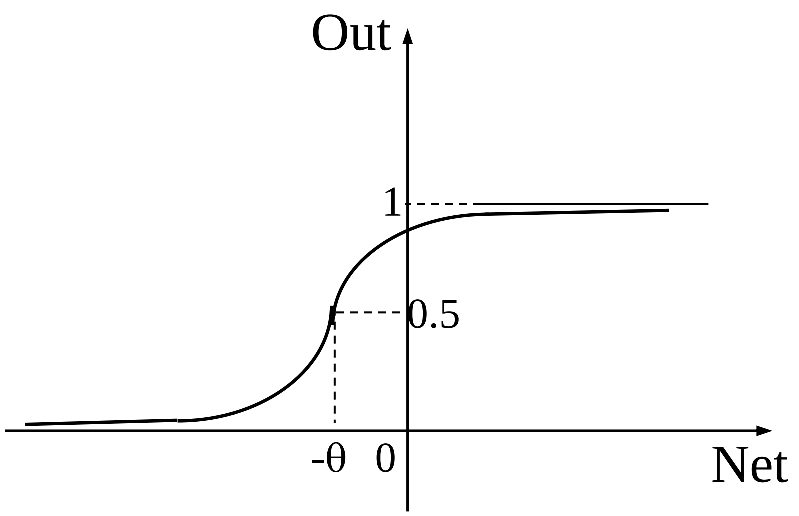


- It is called also saturating linear function
- A combination of linear and hard-limiter activation functions
- α decides the slope in the linear range
- **Disadvantage**: continuous – but not continuously differentiable

Activation function – Sigmoidal

$$Out(Net) = sf(Net, \alpha, \theta) = \frac{1}{1 + e^{-\alpha(Net+\theta)}}$$

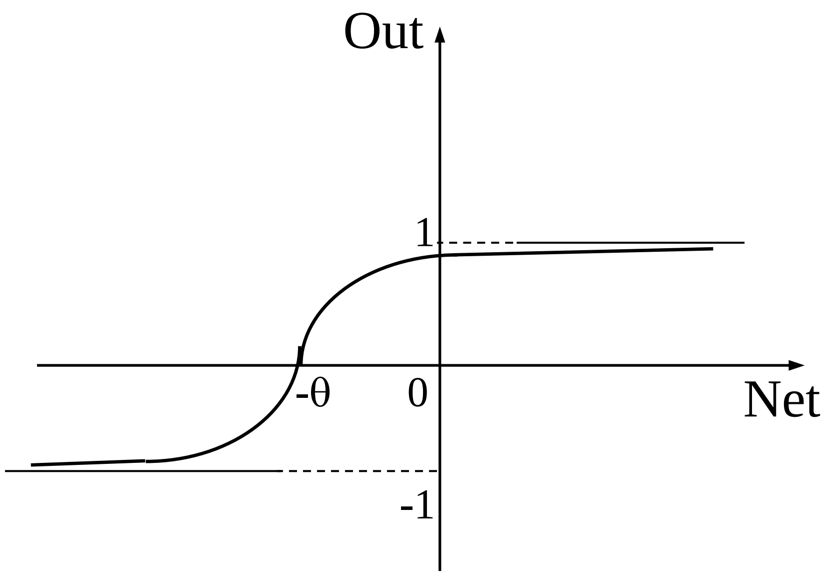
- Most often used in ANNs
- The slope parameter α is important
- The output value is always in (0,1)
- **Advantage**
  - Both continuous and continuously differentiable
  - The derivative of a sigmoidal function can be expressed in terms of the function itself



Activation function – Hyperbolic tangent

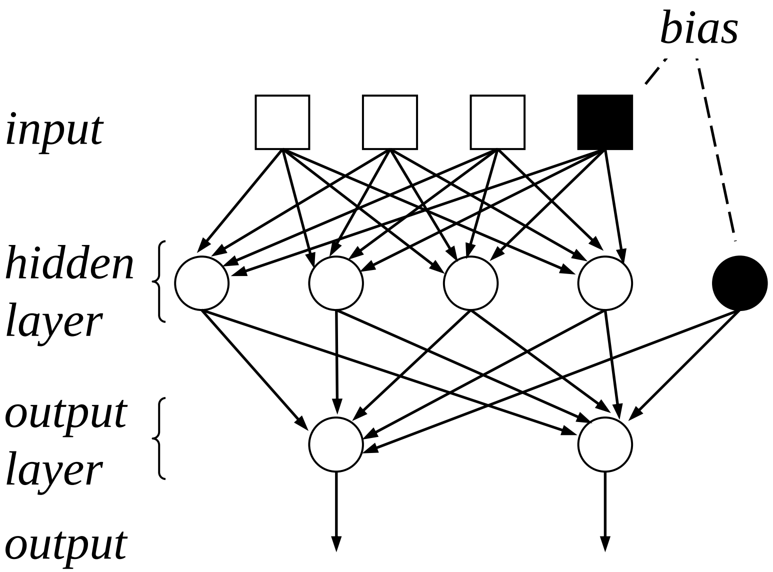
$$Out(Net) = \tanh(Net, \alpha, \theta) = \frac{1 - e^{-\alpha(Net+\theta)}}{1 + e^{-\alpha(Net+\theta)}} = \frac{2}{1 + e^{-\alpha(Net+\theta)}} - 1$$

- Also often used in ANNs
- The slope parameter α is important
- The output value is always in (-1,1)
- **Advantage**
  - Both continuous and continuously differentiable
  - The derivative of a `tanh` function can be expressed in terms of the function itself



Network structure

- Topology of an ANN is composed by:
  - The number of input signals and output signals
  - The number of layers
  - The number of neurons in each layer
  - The number of weights in each neuron
  - The way the weights are linked together within or between the layer(s)
  - Which neurons receive the (error) correction signals
- Every ANN must have
  - exactly one input layer
  - exactly one output layer
  - zero, one, or more than one hidden layer(s)



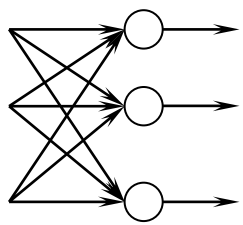
- An ANN with one hidden layer
- Input space: 3-dimensional
- Output space: 2-dimensional
- In total, there are 6 neurons
  - 4 in the hidden layer
  - 2 in the output layer

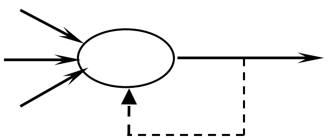
Network structure

- A layer is a group of neurons
- A hidden layer is any layer between the input and the output layers
- Hidden nodes do not directly interact with the external environment
- An ANN is said to be ***fully connected*** if every output from one layer is connected to every node in the next layer
- An ANN is called ***feed-forward network*** if no node output is an input to a node in the same layer or in a preceding layer
- When node outputs can be directed back as inputs to a node in the same (or a preceding) layer, it is a ***feedback network***
  - If the feedback is directed back as input to the nodes in the same layer, then it is called ***lateral feedback***
- Feedback networks that have closed loops are called ***recurrent networks***

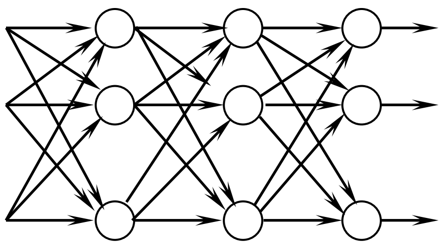
Network structure – Example

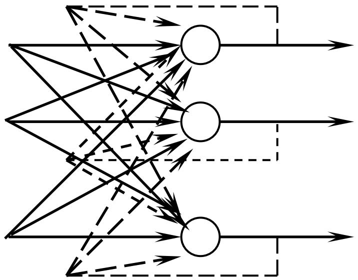
single layer
feed-forward
network





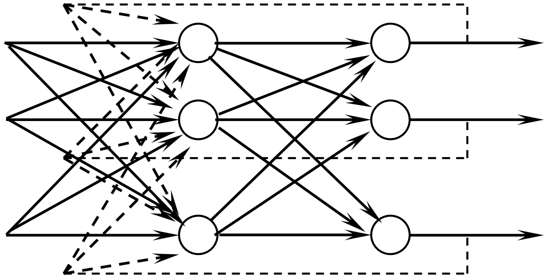
single node with
feedback to itself





single layer
recurrent
network

multilayer
feed-forward
network



multilayer
recurrent
network

Learning rules

- Two kinds of learning in neural networks
  - *Parameter learning*

    → Focus on the update of the connecting weights in an ANN
  - *Structure learning*

    → Focus on the change of the network structure, including the number of processing elements and their connection types
- These two kinds of learning can be performed simultaneously or separately
- Most of the existing learning rules are the type of parameter learning
- We focus the parameter learning

General weight learning rule

- At a learning step (t) the adjustment of the weight vector $\boldsymbol{w}$ is proportional to the product of the learning signal $r^{(t)}$ and the input $\boldsymbol{x}^{(t)}$

$$\Delta\mathbf{w}^{(t)} \sim r^{(t)}.\mathbf{x}^{(t)}$$

$$\Delta\mathbf{w}^{(t)} = \eta.r^{(t)}.\mathbf{x}^{(t)}$$

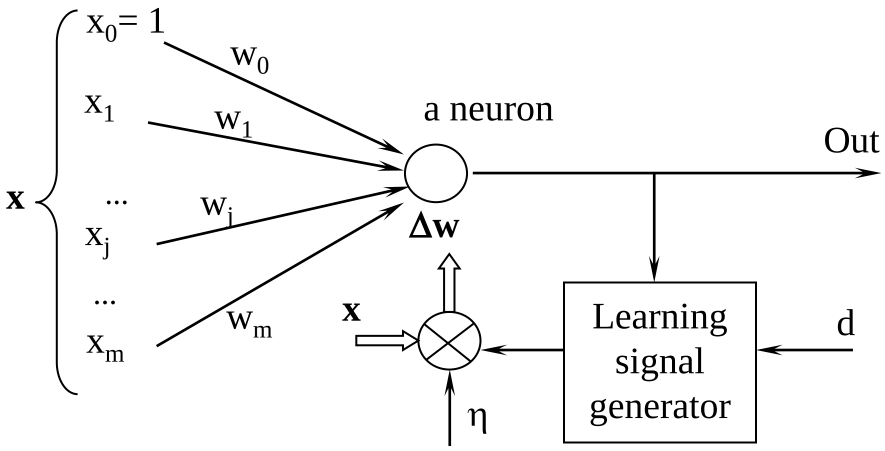
  where η (>0) is the learning rate

- The learning signal r is a function of $\boldsymbol{w}$, $\boldsymbol{x}$, and the desired output d

$$r = g(\mathbf{w},\mathbf{x},d)$$

- The general weight learning rule

$$\Delta\mathbf{w}^{(t)} = \eta.g(\mathbf{w}^{(t)},\mathbf{x}^{(t)},d^{(t)}).\mathbf{x}^{(t)}$$



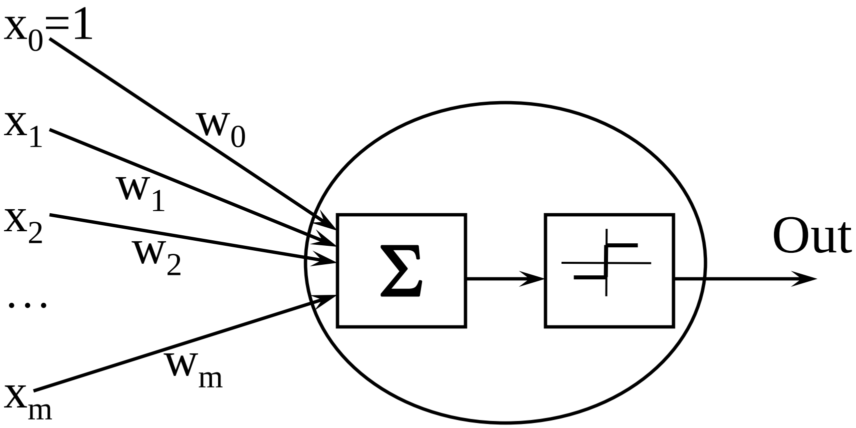
Note that x_j can be either:
- an (external) input signal, or
- an output from another neuron

Perceptron

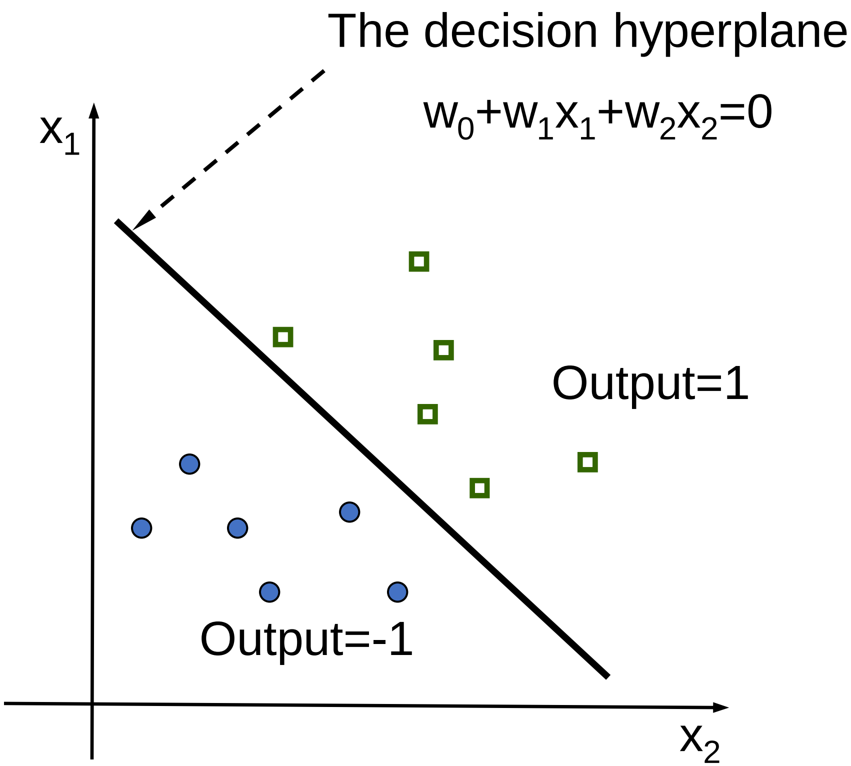
- A perceptron is the simplest type of ANNs
- Use the hard-limit activation function

$$Out = sign\left(Net(w,x)\right) = sign\left(\sum_{j=0}^{m} w_j x_j\right)$$

- For an instance ***x***, the perceptron output is
  - 1, if Net(**w**,**x**)>0
  - -1, otherwise



Perceptron – Illustration



Perceptron – Learning

- Given a training set D= {(**x**,d)}
  - ***x*** is the input vector
  - *d* is the desired output value (i.e., -1 or 1)
- The perceptron learning is to determine a weight vector that makes the perceptron produce the correct output (-1 or 1) for every training instance
- If a training instance ***x*** is correctly classified, then no update is needed
- If d=1 but the perceptron outputs -1, then the weight ***w*** should be updated so that Net(**w**,**x**) is increased
- If d=-1 but the perceptron outputs 1, then the weight ***w*** should be updated so that Net(**w**,**x**) is decreased

```
Perceptron_incremental(D, η)
Initialize w  (w_i ← an initial (small) random value)
do
    for each training instance (x,d) ∈ D
        Compute the real output value Out
        if (Out≠d)
            w ← w + η(d-Out)x
    end for
until all the training instances in D are correctly classified
return w
```

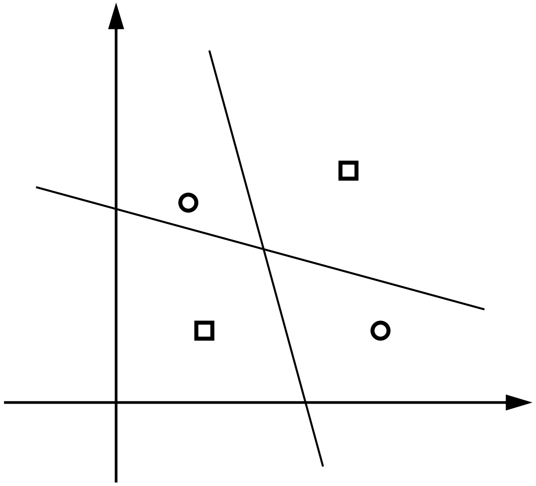
Perceptron_batch(D, η)

```
Initialize w  (w_i ← an initial (small) random value)
do
    Δw ← 0
    for each training instance (x,d) ∈ D
        Compute the real output value Out
        if (Out≠d)
            Δw ← Δw + η(d-Out)x
    end for
    w ← w + Δw
until all the training instances in D are correctly classified
return w
```

Perceptron - Limitation

- The perceptron learning procedure is proven to converge if
  - The training instances are linearly separable
  - With a sufficiently small η used
- The perceptron may not converge if the training instances are not linearly separable
- We need to use the **delta rule**
  - Converges toward a best-fit approximation of the target function
  - The delta rule uses **gradient descent** to search the hypothesis space (of possible weight vectors) to find the weight vector that best fits the training instances

A perceptron cannot correctly classify this training set!



Error (cost) function

- Let's consider an ANN that has *n* output neurons
- Given a training instance (**x**,d), the **training error** made by the currently estimated weights vector **w**:

$$E_{\mathbf{x}}(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}\left(d_i - Out_i\right)^2$$

- The **training error** made by the currently estimated weights vector **w** over the entire training set *D*:

$$E_D(\mathbf{w}) = \frac{1}{|D|}\sum_{\mathbf{x}\in D} E_{\mathbf{x}}(\mathbf{w})$$

Gradient descent

- **Gradient** of E (denoted as ∇E) is a vector
  - The direction points most uphill
  - The length is proportional to steepness of hill
- The gradient of ∇E specifies the direction that produces the **steepest increase** in E

$$\nabla E(\mathbf{w}) = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_N} \right)$$

where N is the number of the weights in the network (i.e., N is the length of $\mathbf{w}$)

- Hence, the direction that produces the **steepest decrease** is the negative of the gradient of E

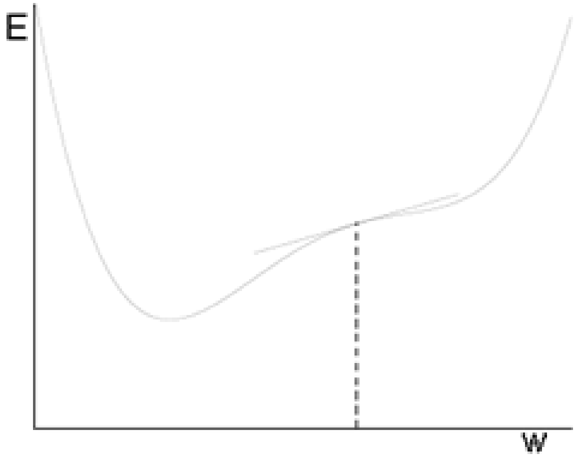
$$\Delta \mathbf{w} = -\eta \, . \nabla \mathrm{E}\,(\mathbf{w})\,; \qquad \Delta w_i = -\eta \frac{\partial E}{\partial w_i}, \quad \forall i = 1..N$$

- Requirement: The activation functions used in the network must be continuous functions of the weights, differentiable everywhere

Gradient descent – Illustration

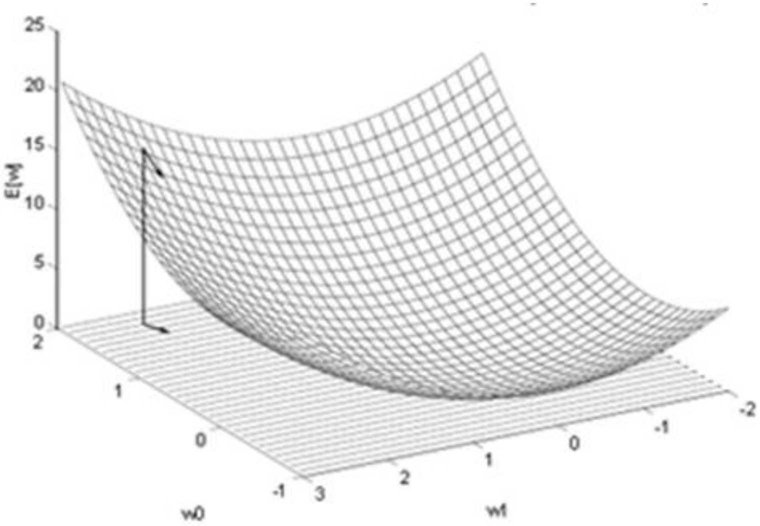
One-dimensional

E(w)



Two-dimensional

$E(w_1,w_2)$



Gradient_descent_incremental (D, η)

Initialize $\mathbf{w}$ ($w_i \leftarrow$ an initial (small) random value)

do

for each training instance $(\mathbf{x}, d) \in D$

Compute the network output

for each weight component w_i

$$w_i \leftarrow w_i - \eta(\partial E_{\mathbf{x}} / \partial w_i)$$

end for

end for

until (stopping criterion satisfied)

return $\mathbf{w}$

Stopping criterion: # of iterations (epochs), threshold error, etc.

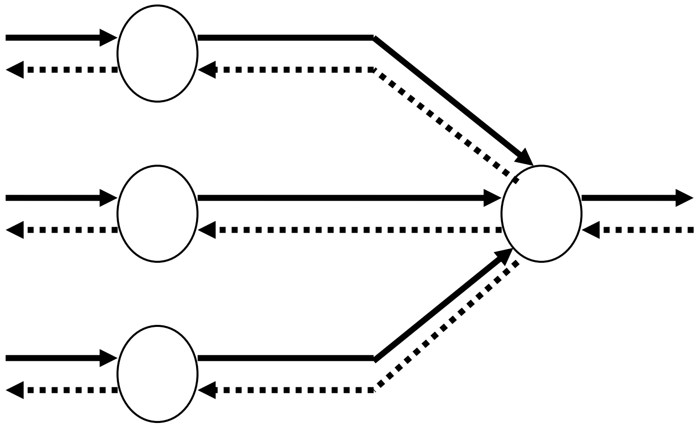
Multi-layer NNs and Back-propagation alg.

- 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 (1)

- 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
  - **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)
  - **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

Back-propagation algorithm (2)

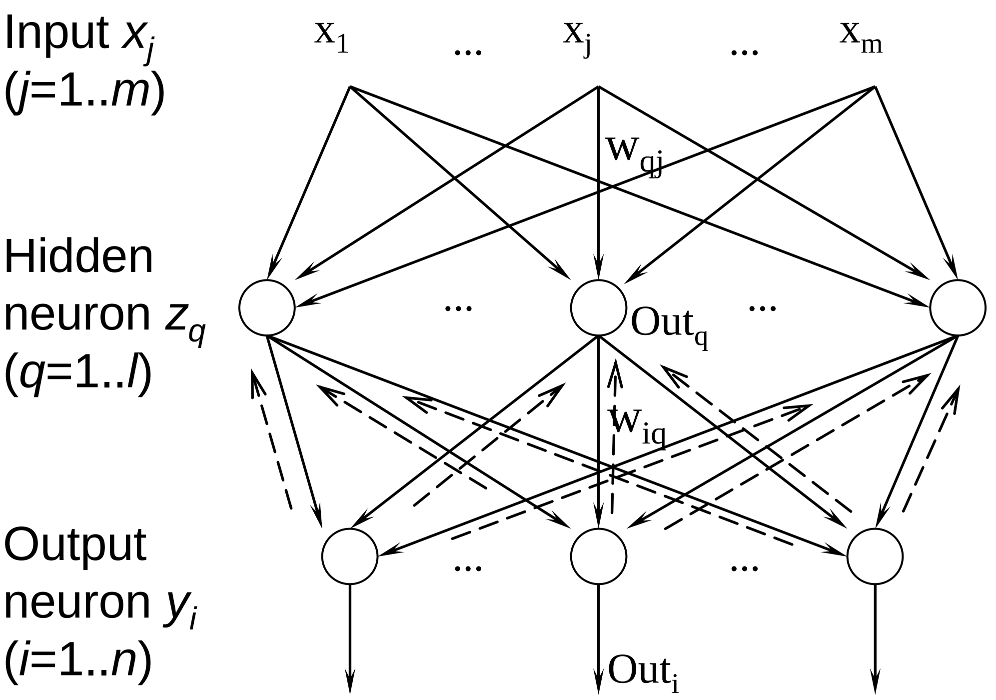


Signal forward phase
- Network activation

Error backward phase
- Output error computation
- Error propagation

Derivation of BP alg. – Network structure

- 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



BP algorithm – Forward phase (1)

- 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

BP algorithm – Forward phase (2)

- 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 $\boldsymbol{x}$

BP algorithm – Backward phase (1)

- 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

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

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

BP algorithm – Backward phase (2)

- 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 $f'(\text{Net}_i)=\partial f(\text{Net}_i)/\partial \text{Net}_i$

BP algorithm – Backward phase (3)

- 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 $E(\mathbf{w})$, it is clear that each error term $(d_i - y_i)$ $(i=1..n)$ is a function of Out_q

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

BP algorithm – Backward phase (4)

- 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 $f'(\mathrm{Net}_q) = \partial f(\mathrm{Net}_q) / \partial \mathrm{Net}_q$

BP algorithm – Backward phase (5)

- According to the error equations δ_i and δ_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** δ_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!

BP algorithm – Backward phase (6)

- 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 $(q\text{-}1)^{th}$ 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

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

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

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)

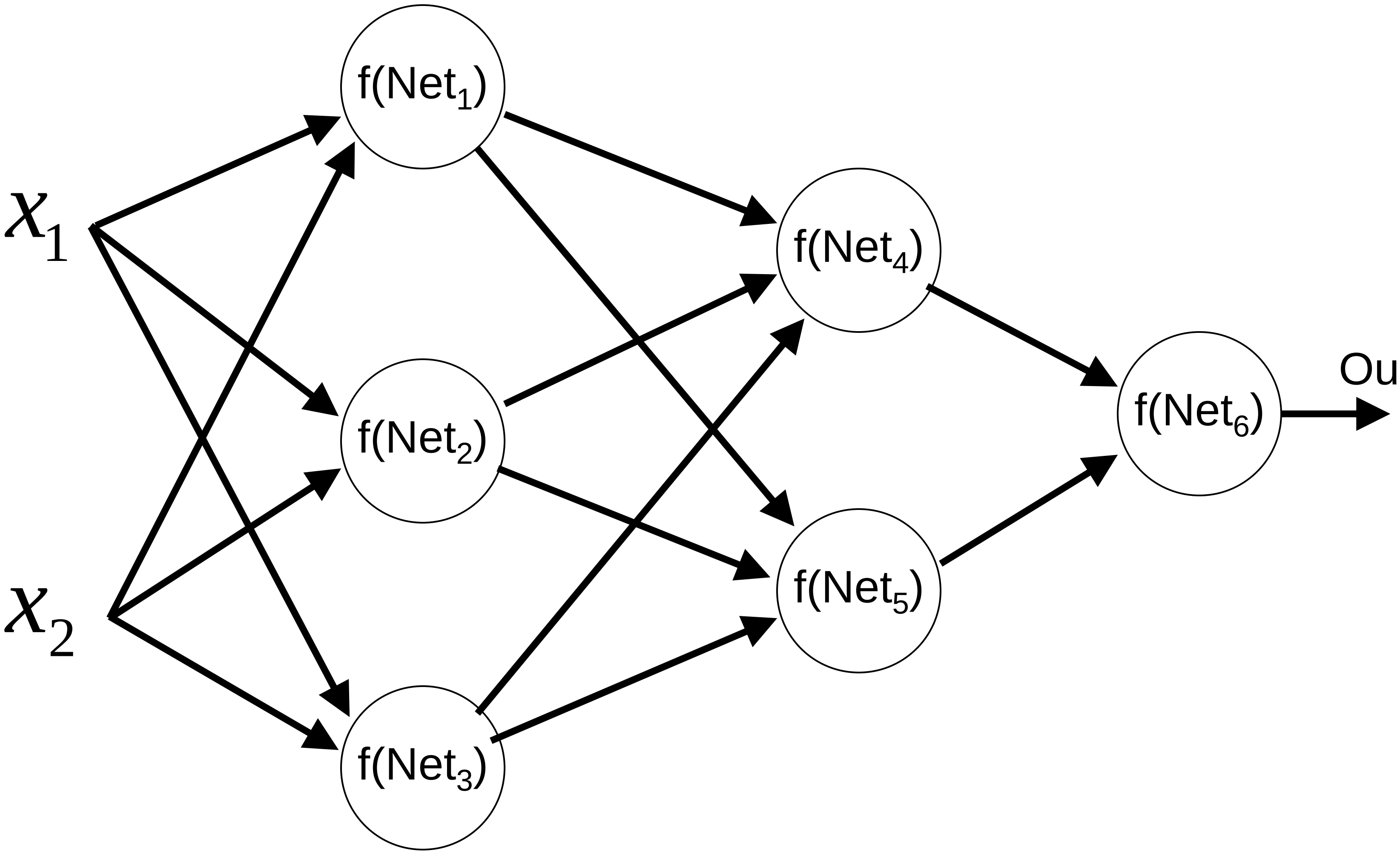
If the entire training set has been exploited, then go to step 6; otherwise, go to step 1

Step 6 (Total error check)

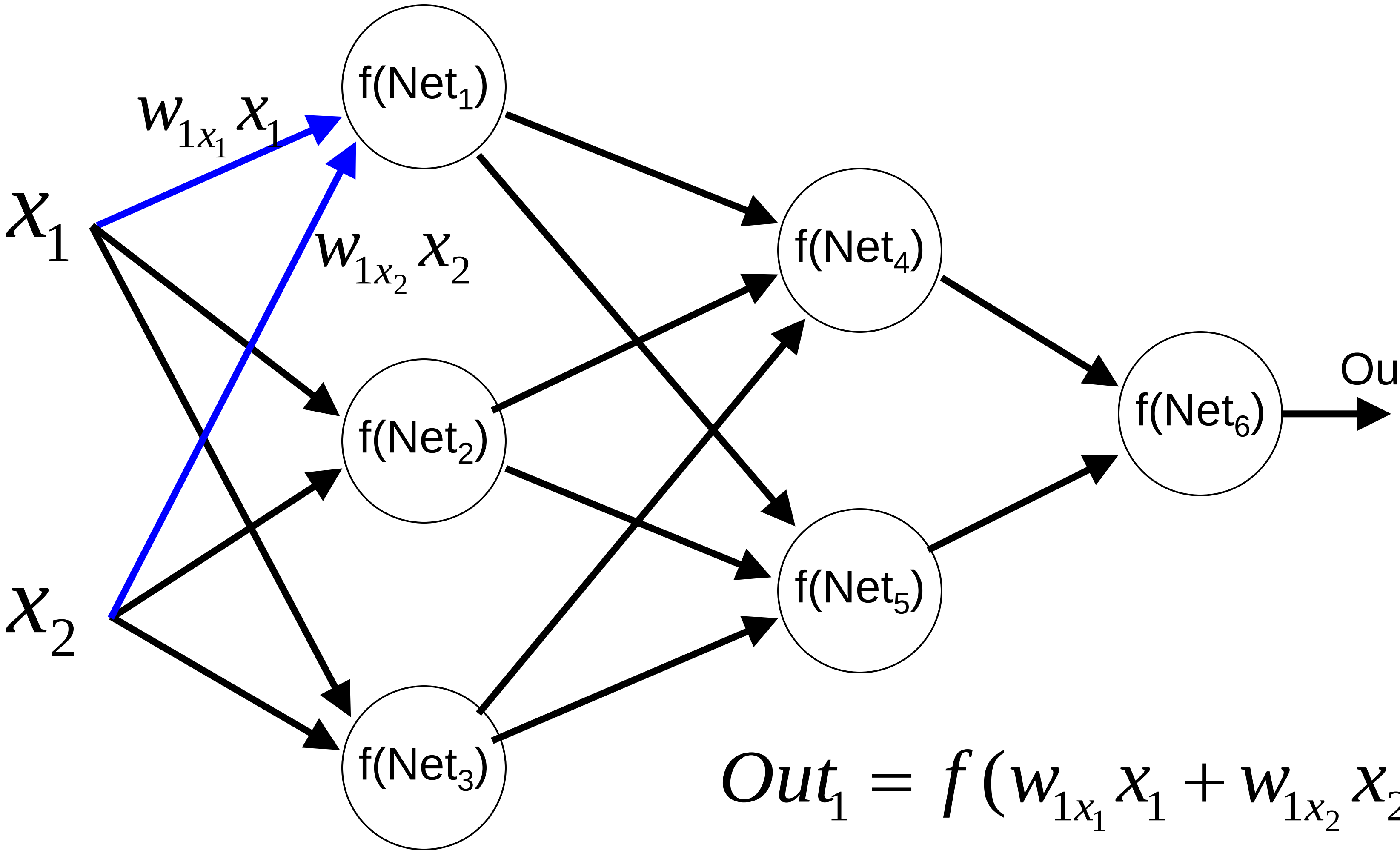
If the current total error is acceptable ($E<E_{threshold}$) then the training process terminates and output the final weights;

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

BP illustration – Forward phase (1)

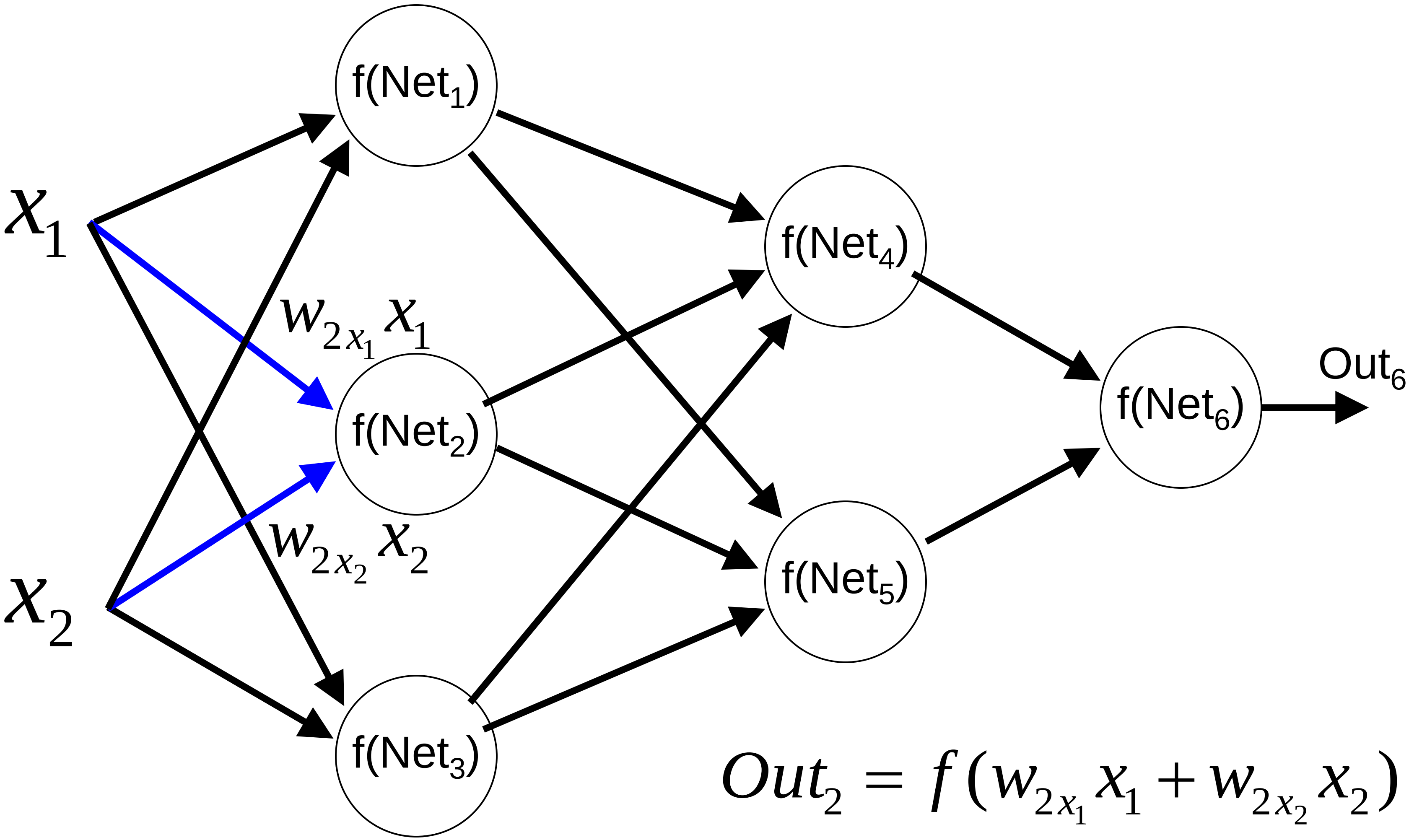


BP illustration – Forward phase (2)



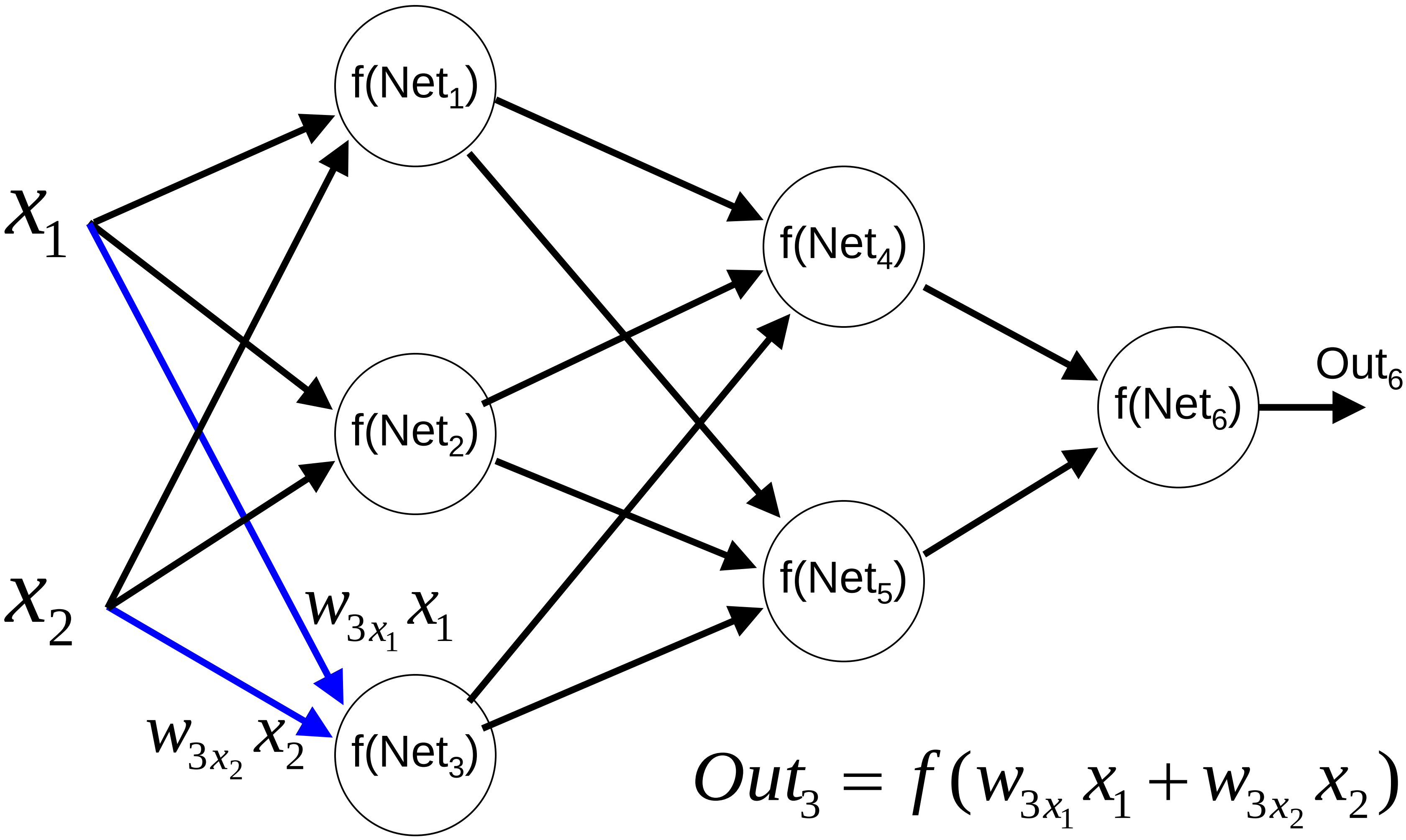
$$Out_1 = f(w_{1x_1}x_1 + w_{1x_2}x_2)$$

BP illustration – Forward phase (3)



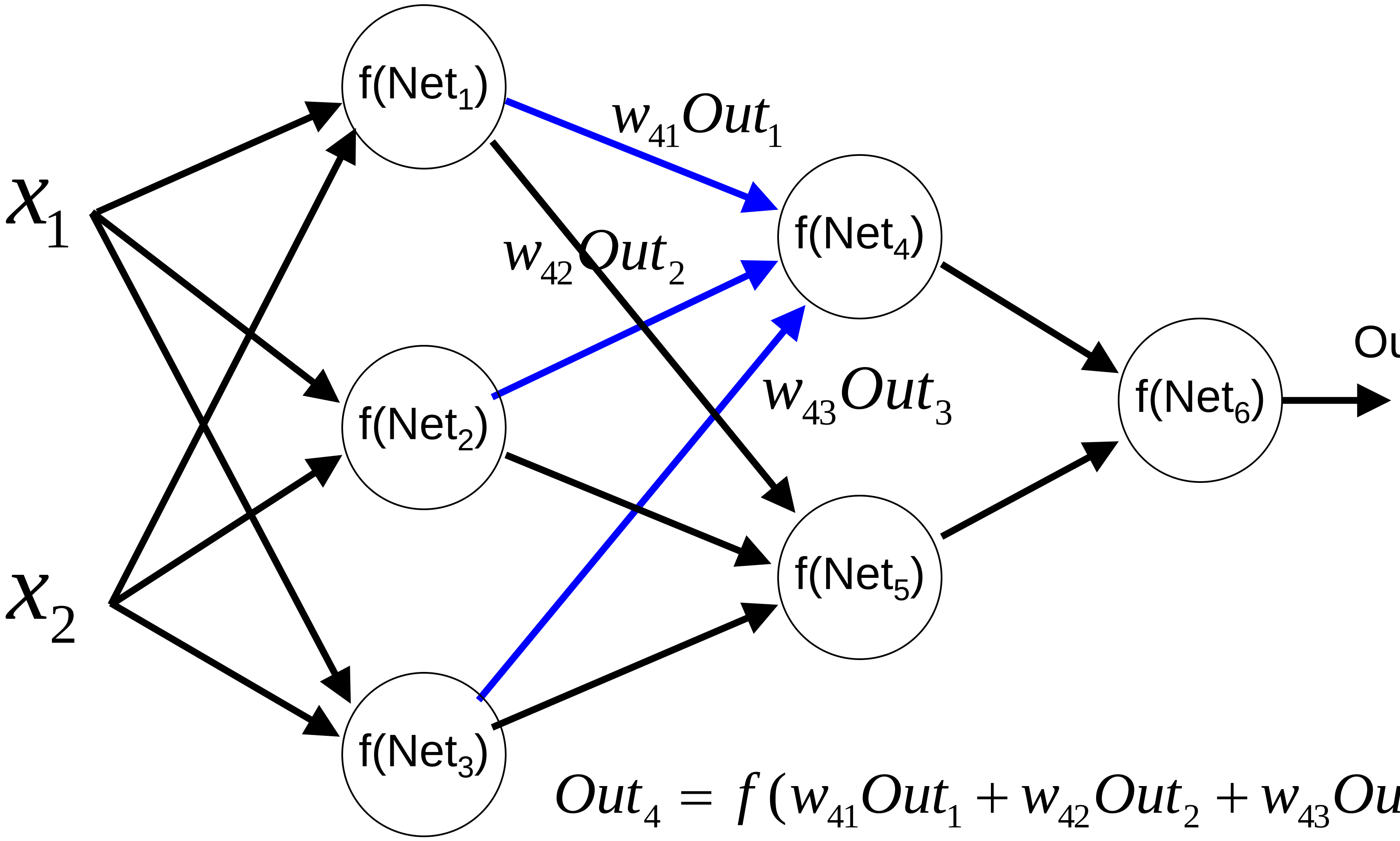
$$Out_2 = f(w_{2x_1}x_1 + w_{2x_2}x_2)$$

BP illustration – Forward phase (4)



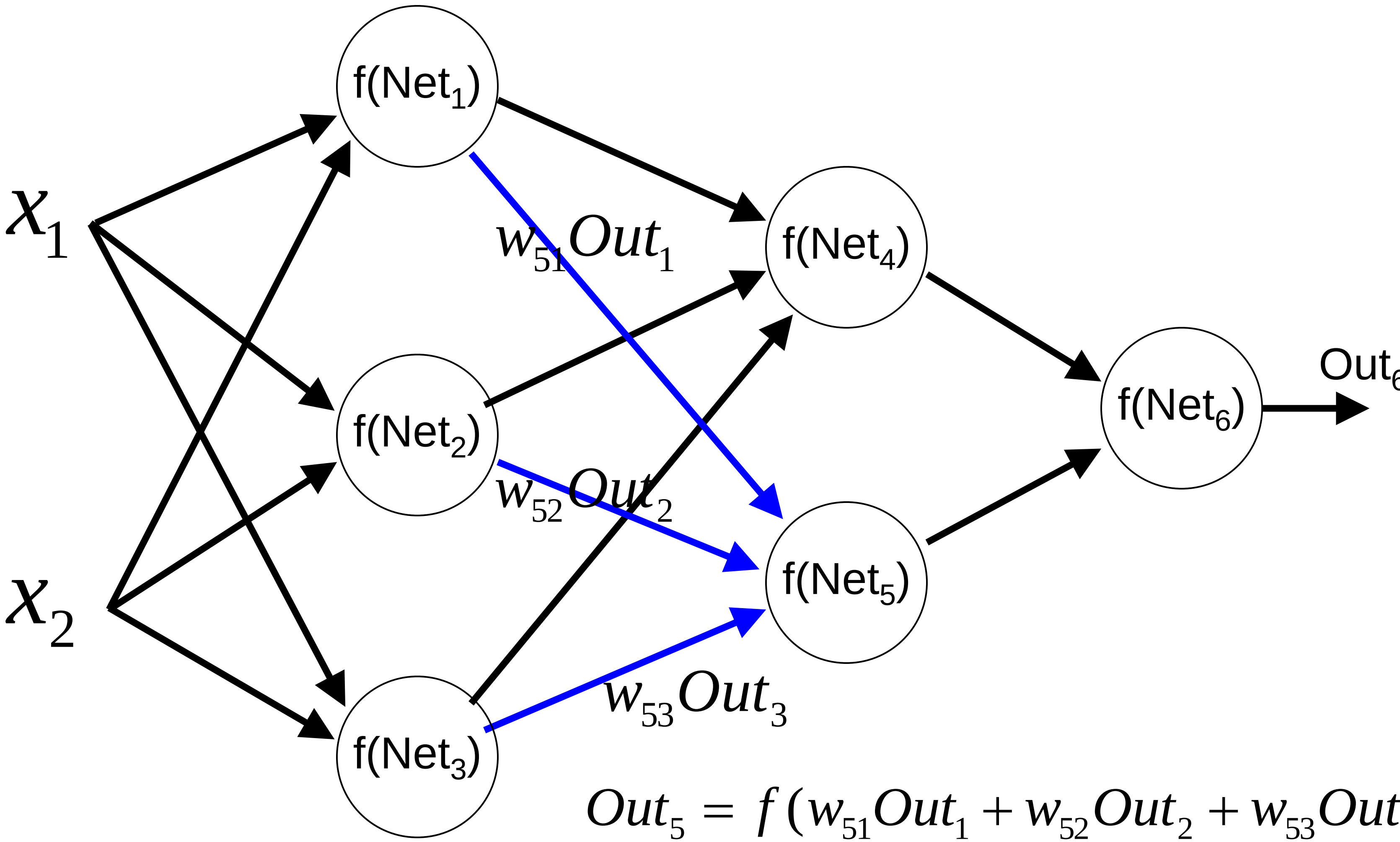
$$Out_3 = f\left(w_{3x_1}x_1 + w_{3x_2}x_2\right)$$

BP illustration – Forward phase (5)



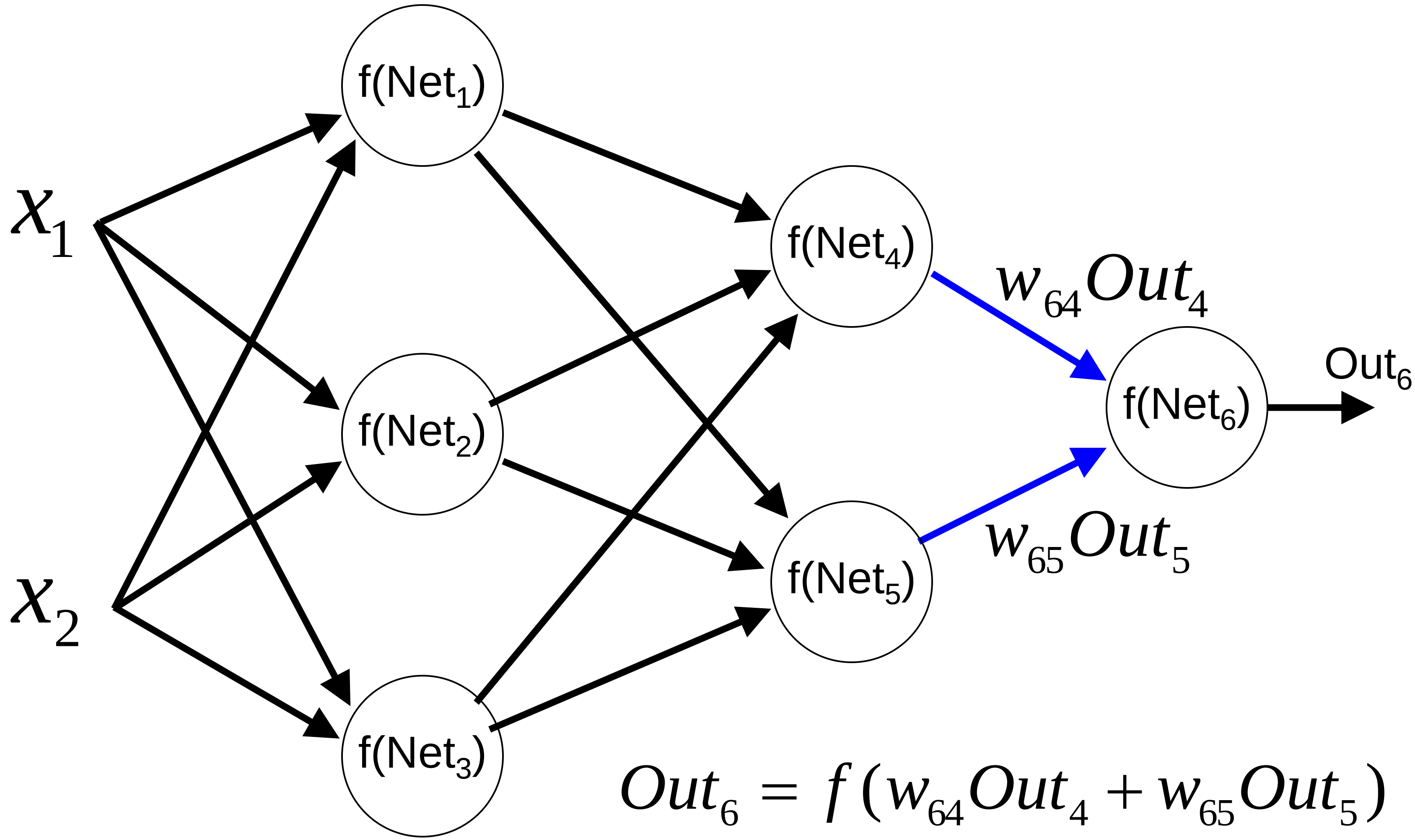
$$Out_4 = f(w_{41}Out_1 + w_{42}Out_2 + w_{43}Out_3)$$

BP illustration – Forward phase (6)



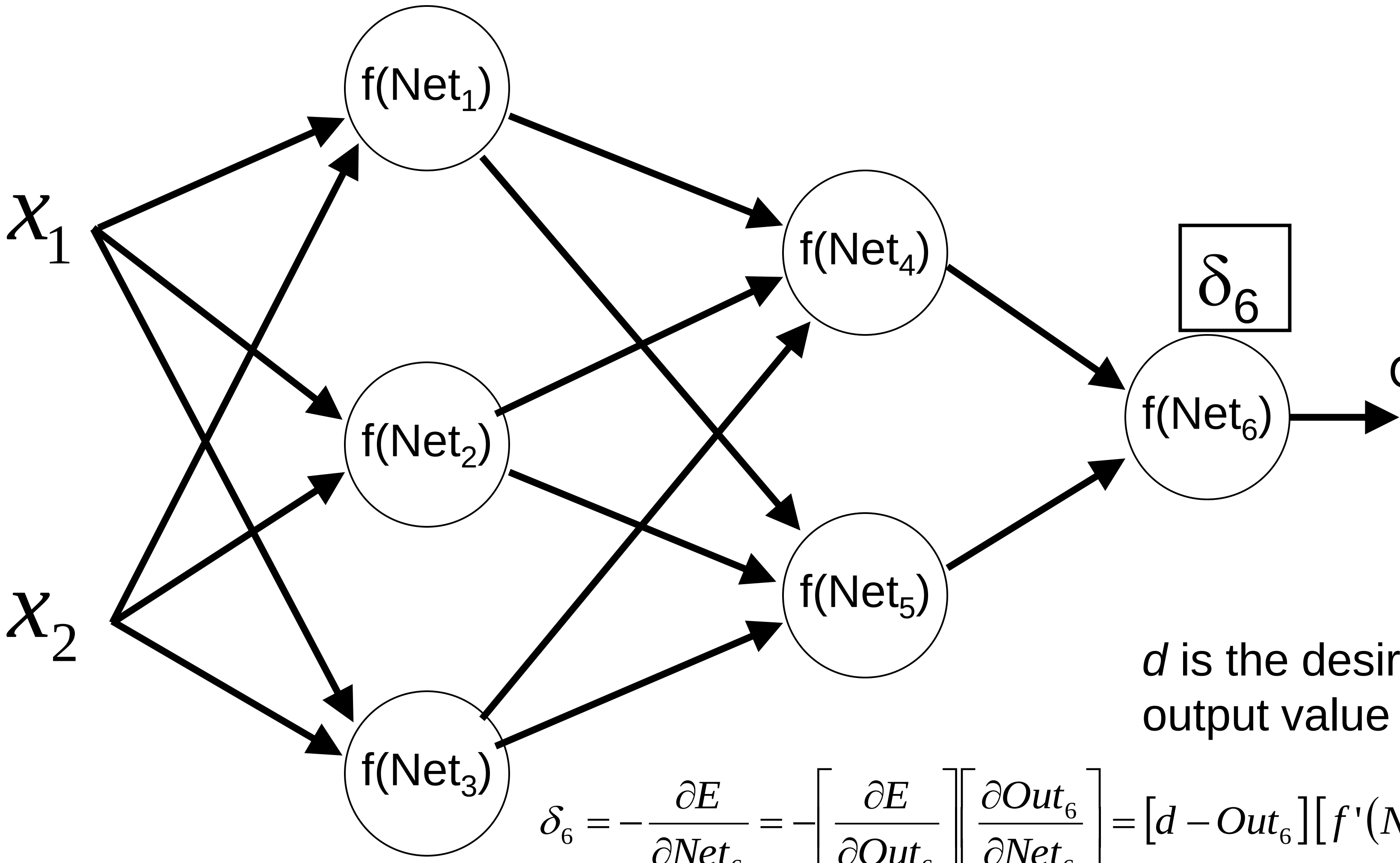
$$Out_5 = f(w_{51}Out_1 + w_{52}Out_2 + w_{53}Out_3)$$

BP illustration – Forward phase (7)



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

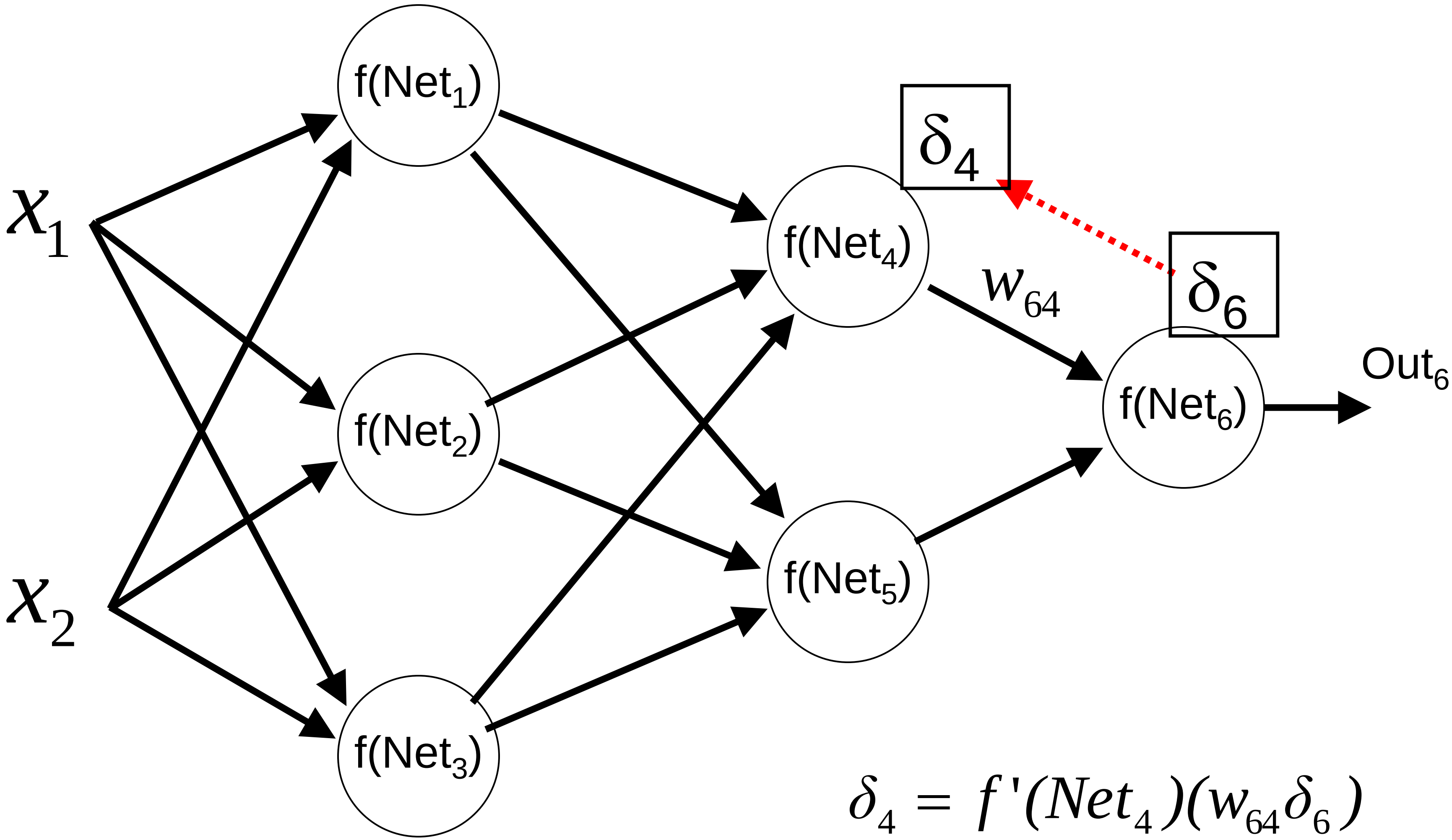
BP illustration – Compute the error



d is the desired output value

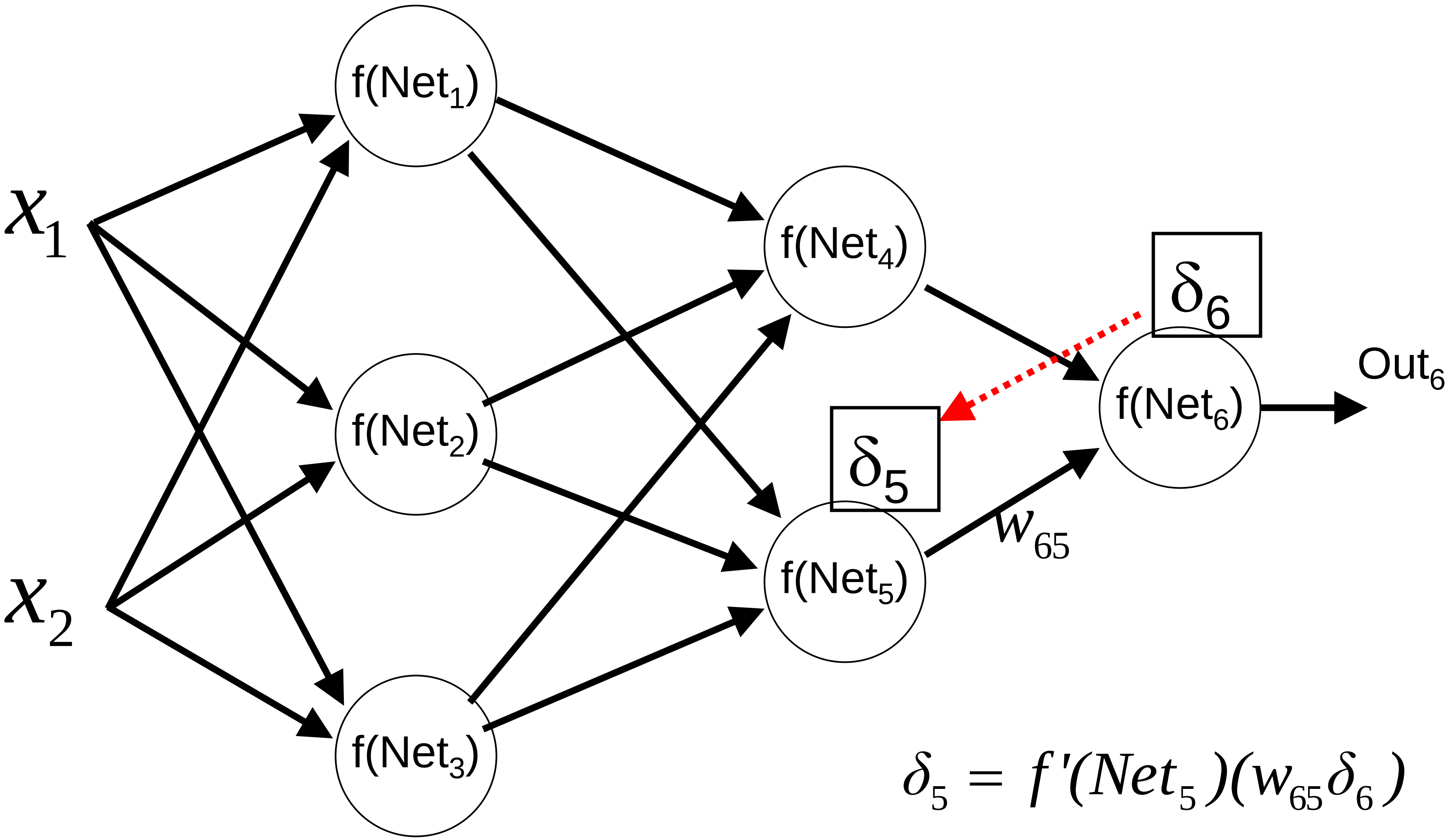
$$\delta_6 = -\frac{\partial E}{\partial Net_6} = -\left[\frac{\partial E}{\partial Out_6}\right]\left[\frac{\partial Out_6}{\partial Net_6}\right] = [d - Out_6][f'(Net_6)]$$

BP illustration – Backward phase (1)



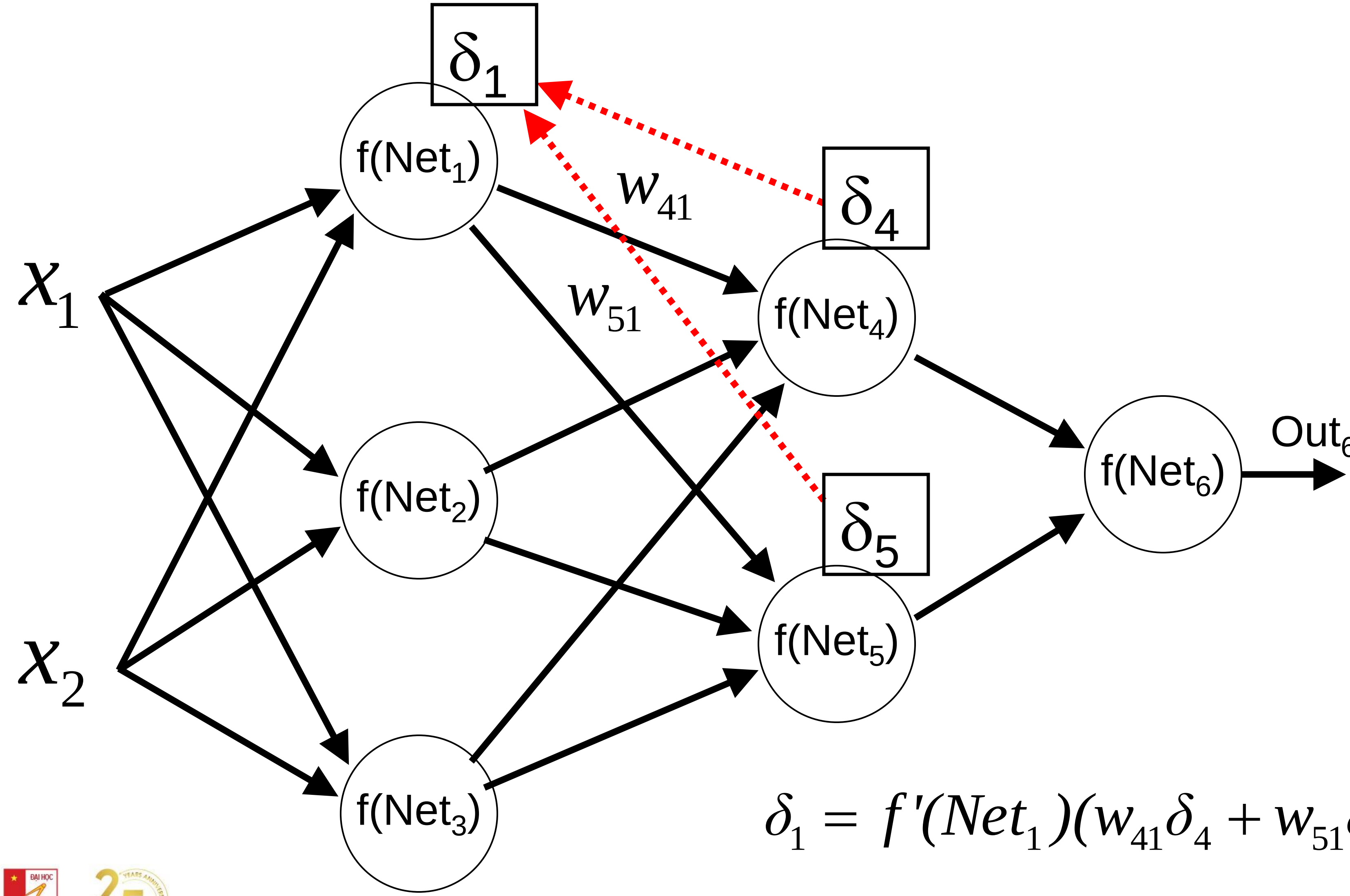
$$\delta_4 = f'(Net_4)(w_{64}\delta_6)$$

BP illustration – Backward phase (2)



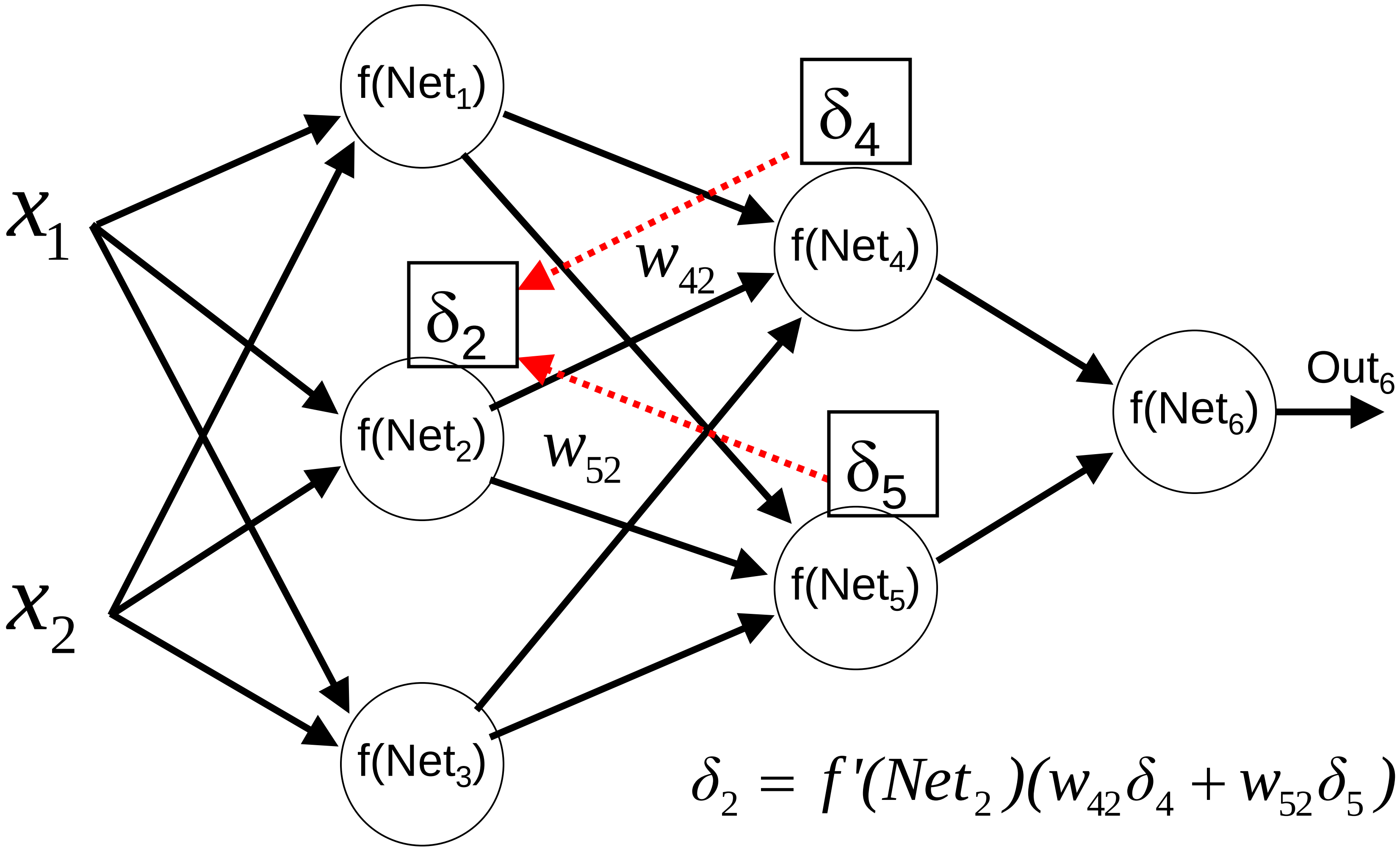
$$\delta_5 = f'(Net_5)(w_{65}\delta_6)$$

BP illustration – Backward phase (3)



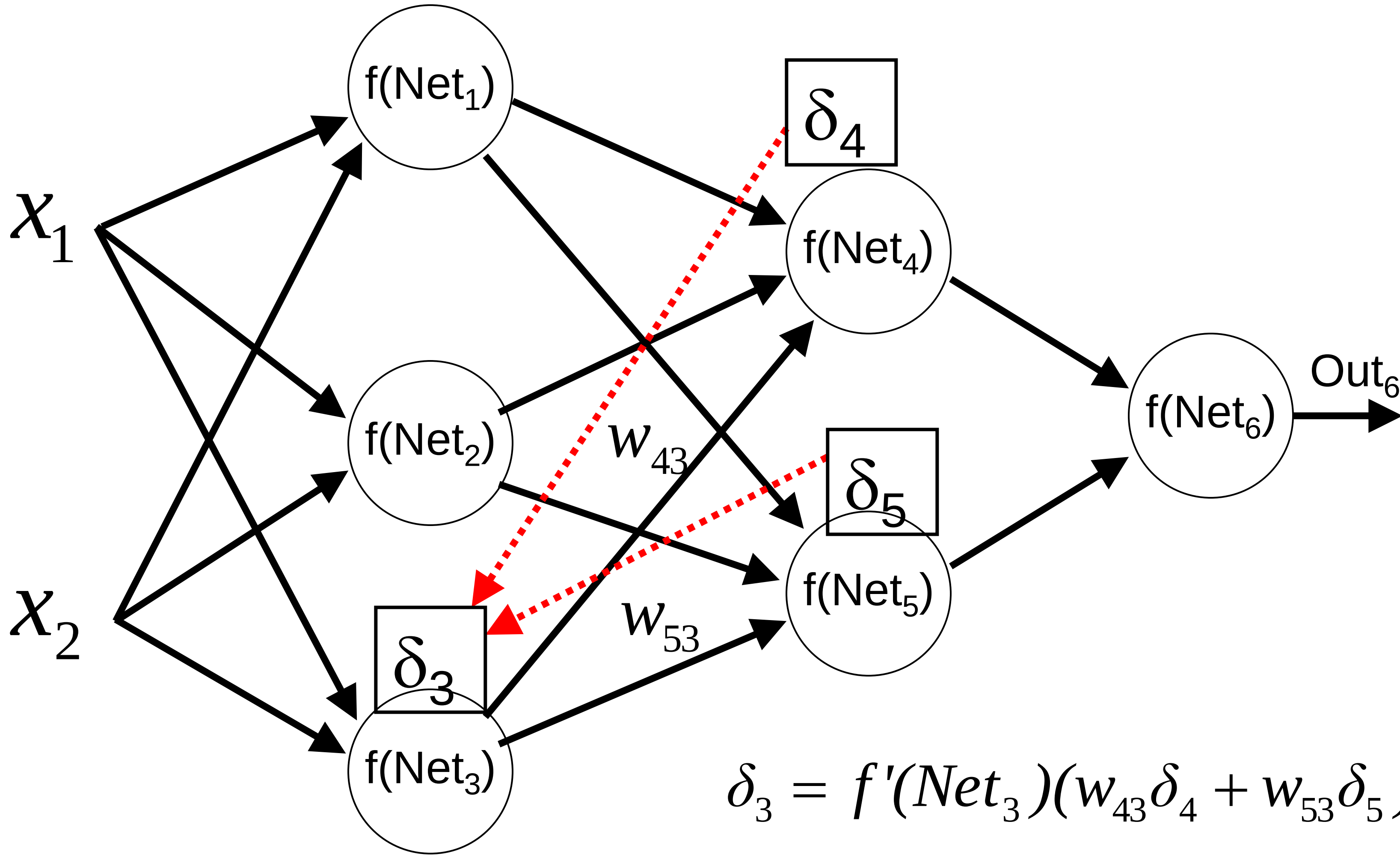
$$\delta_1 = f'(Net_1)(w_{41}\delta_4 + w_{51}\delta_5)$$

BP illustration – Backward phase (4)



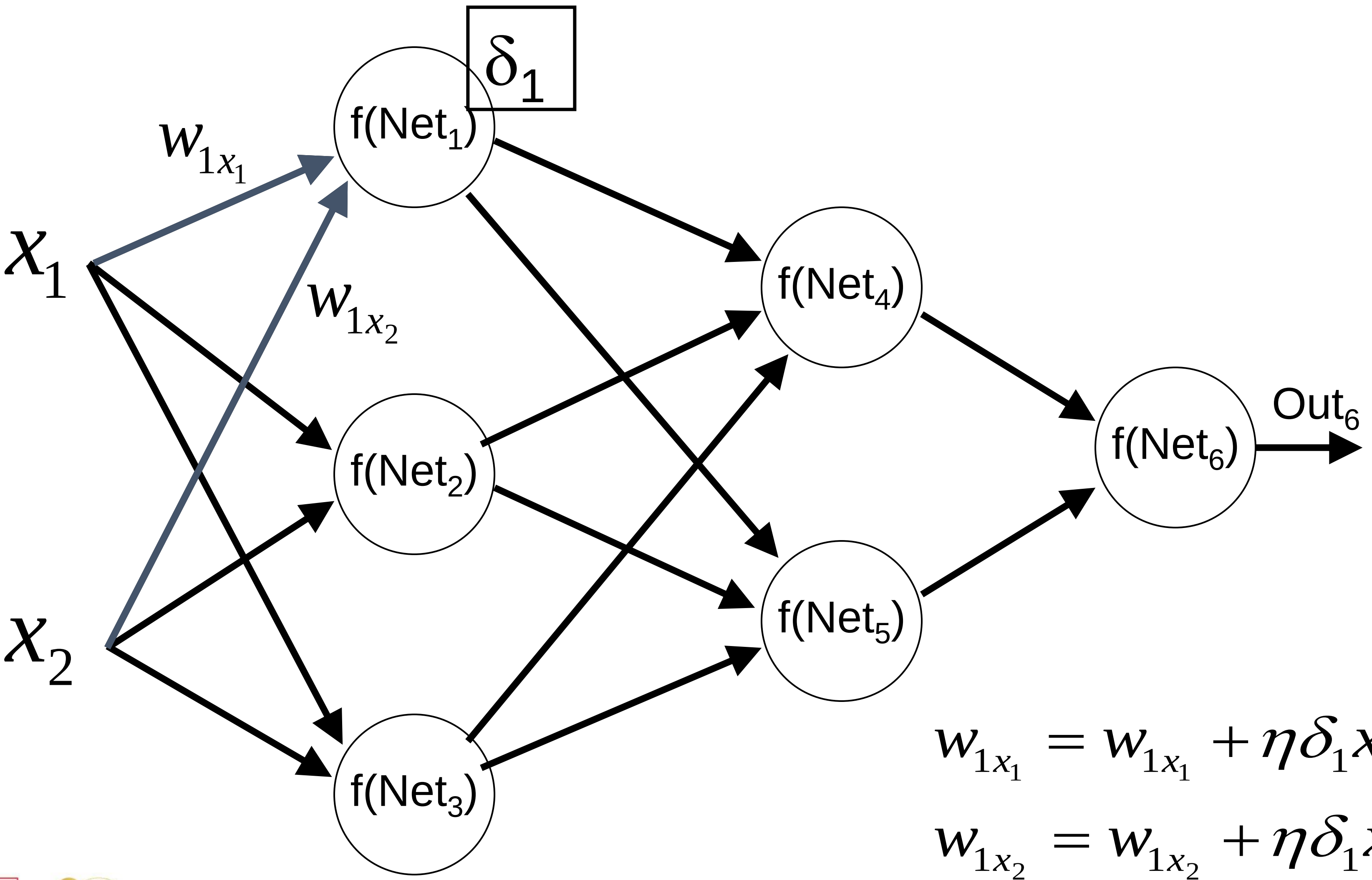
$$\delta_2 = f'(Net_2)(w_{42}\delta_4 + w_{52}\delta_5)$$

BP illustration – Backward phase (5)



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

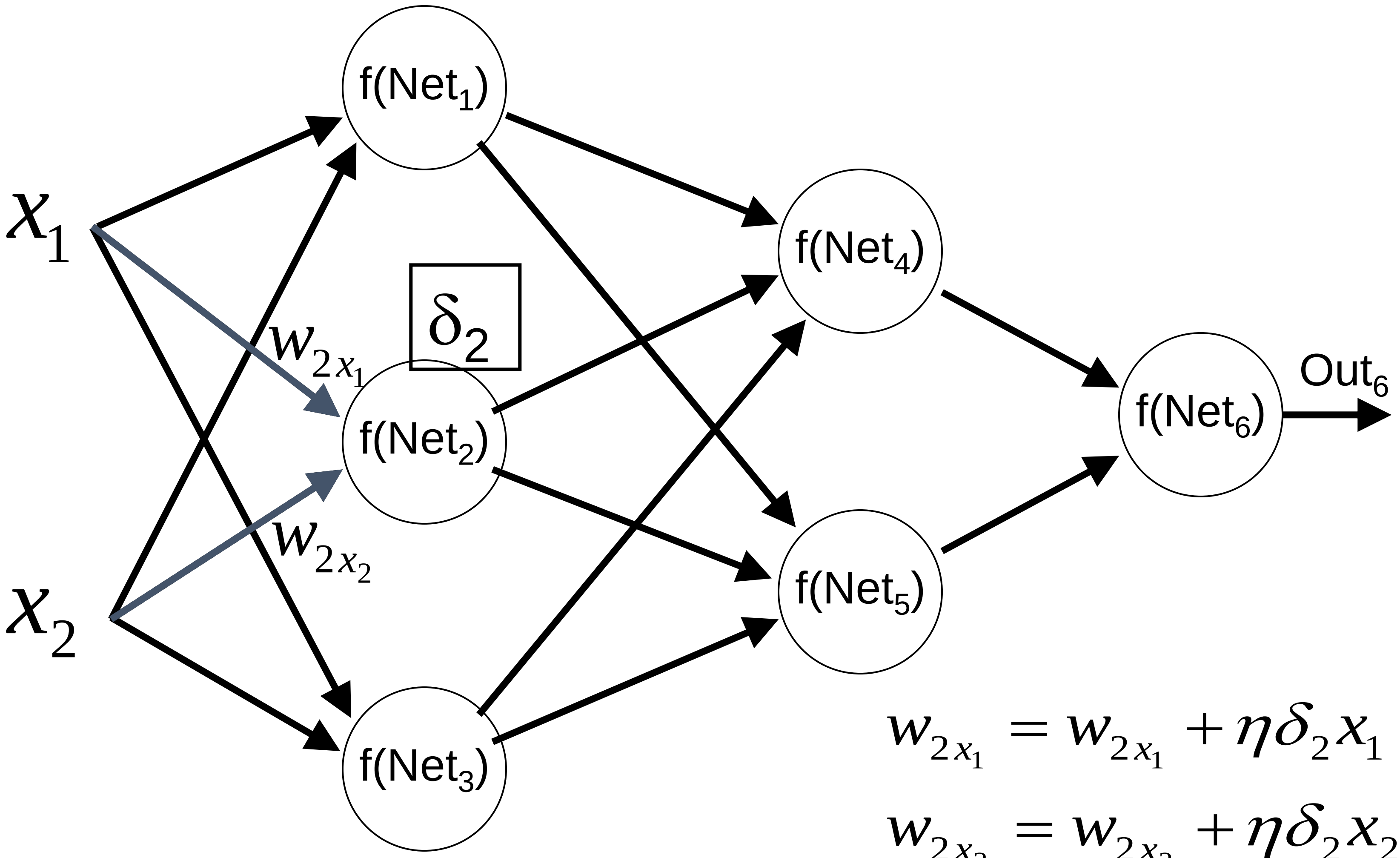
BP illustration – Weight update (1)



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

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

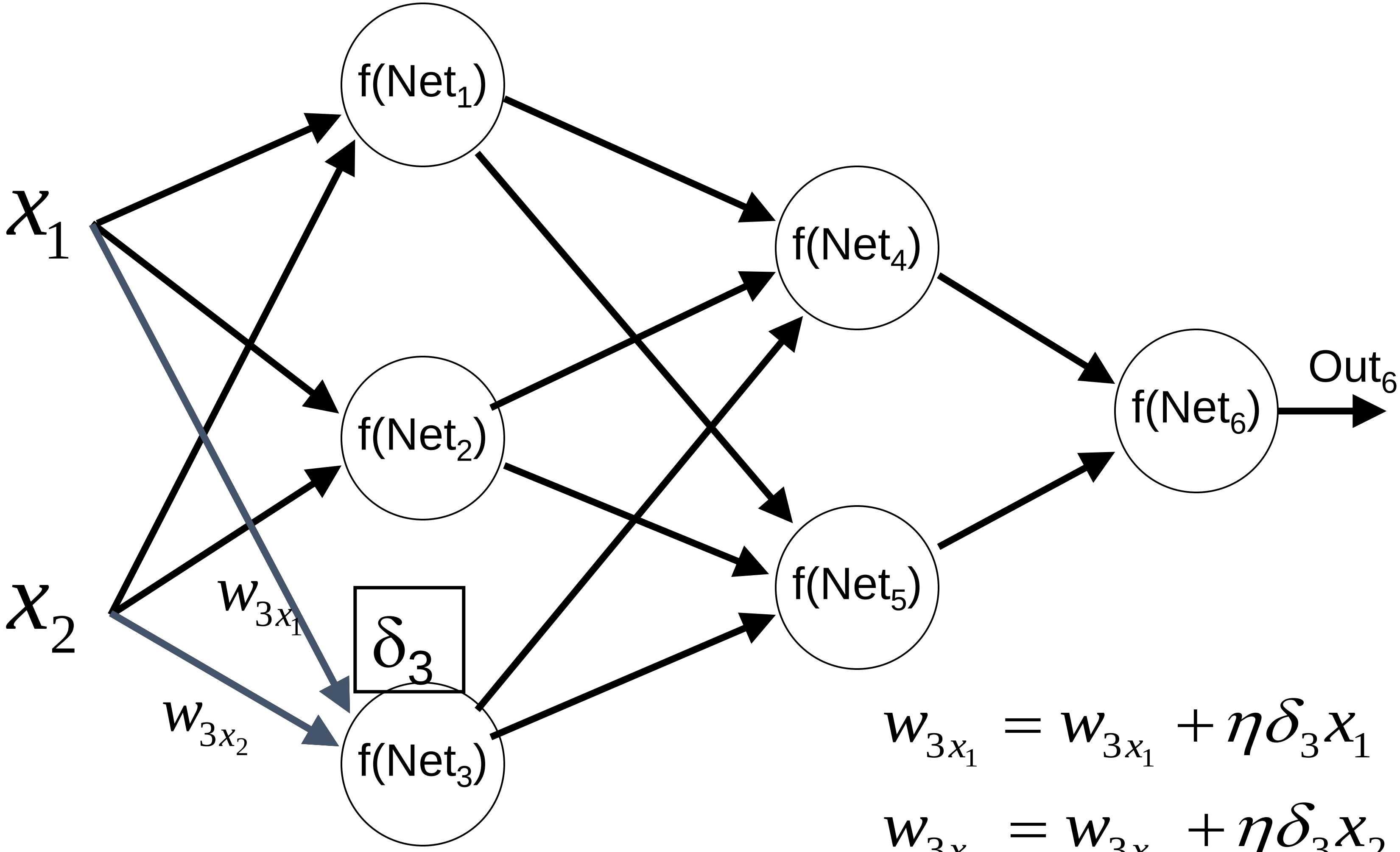
BP illustration – Weight update (2)



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

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

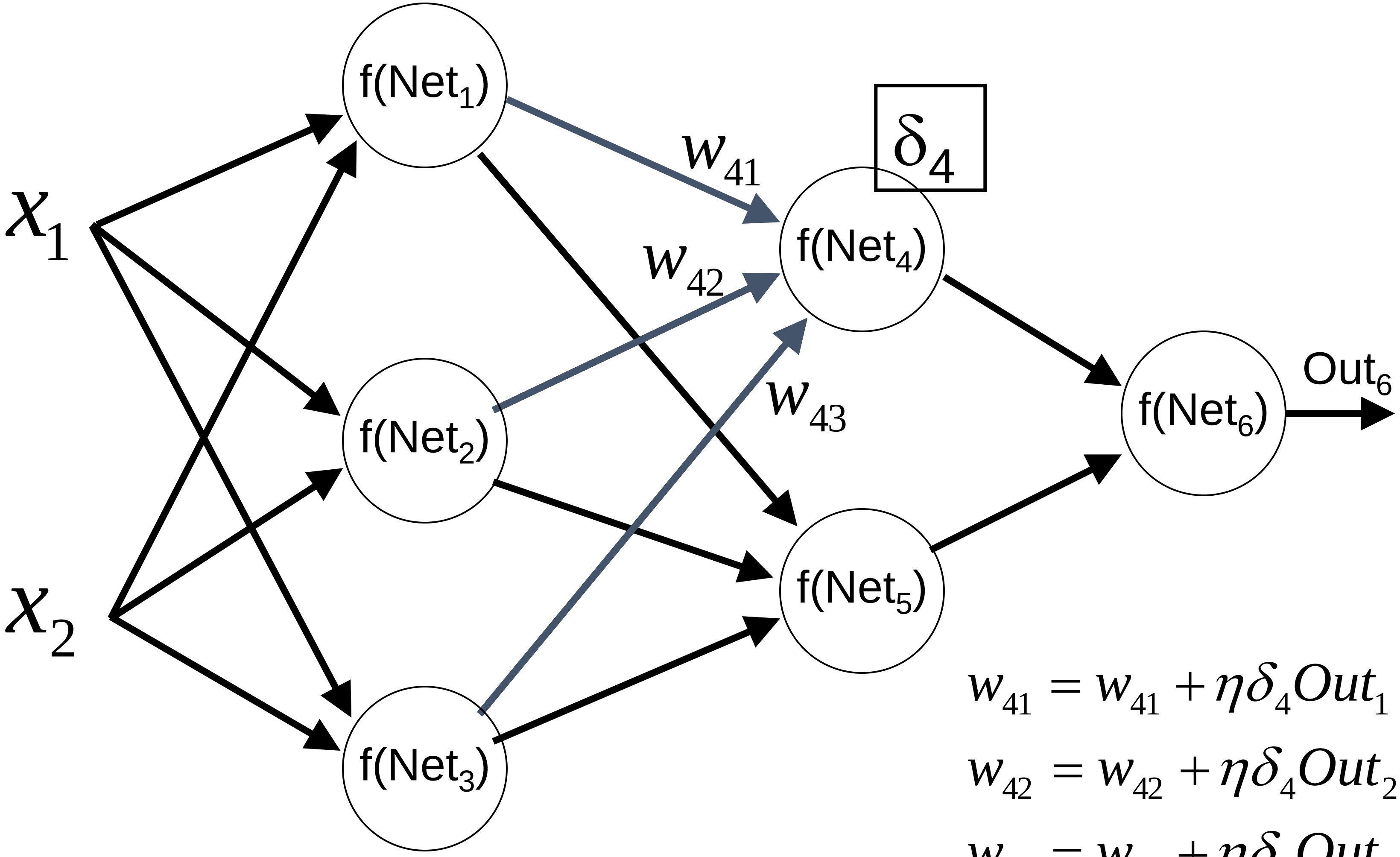
BP illustration – Weight update (3)



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

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

BP illustration – Weight update (4)

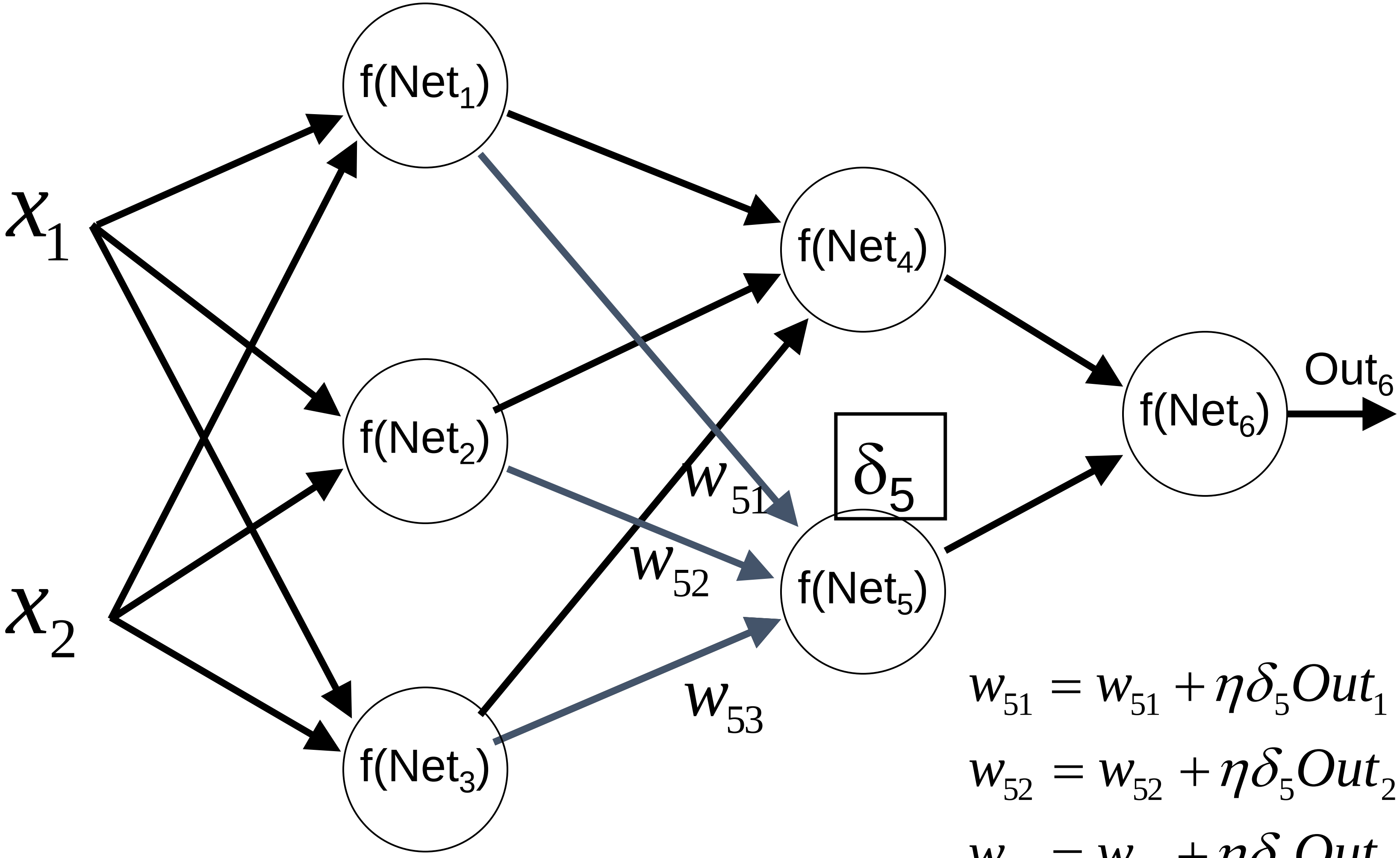


$$w_{41} = w_{41} + \eta\delta_4 Out_1$$

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

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

BP illustration – Weight update (5)

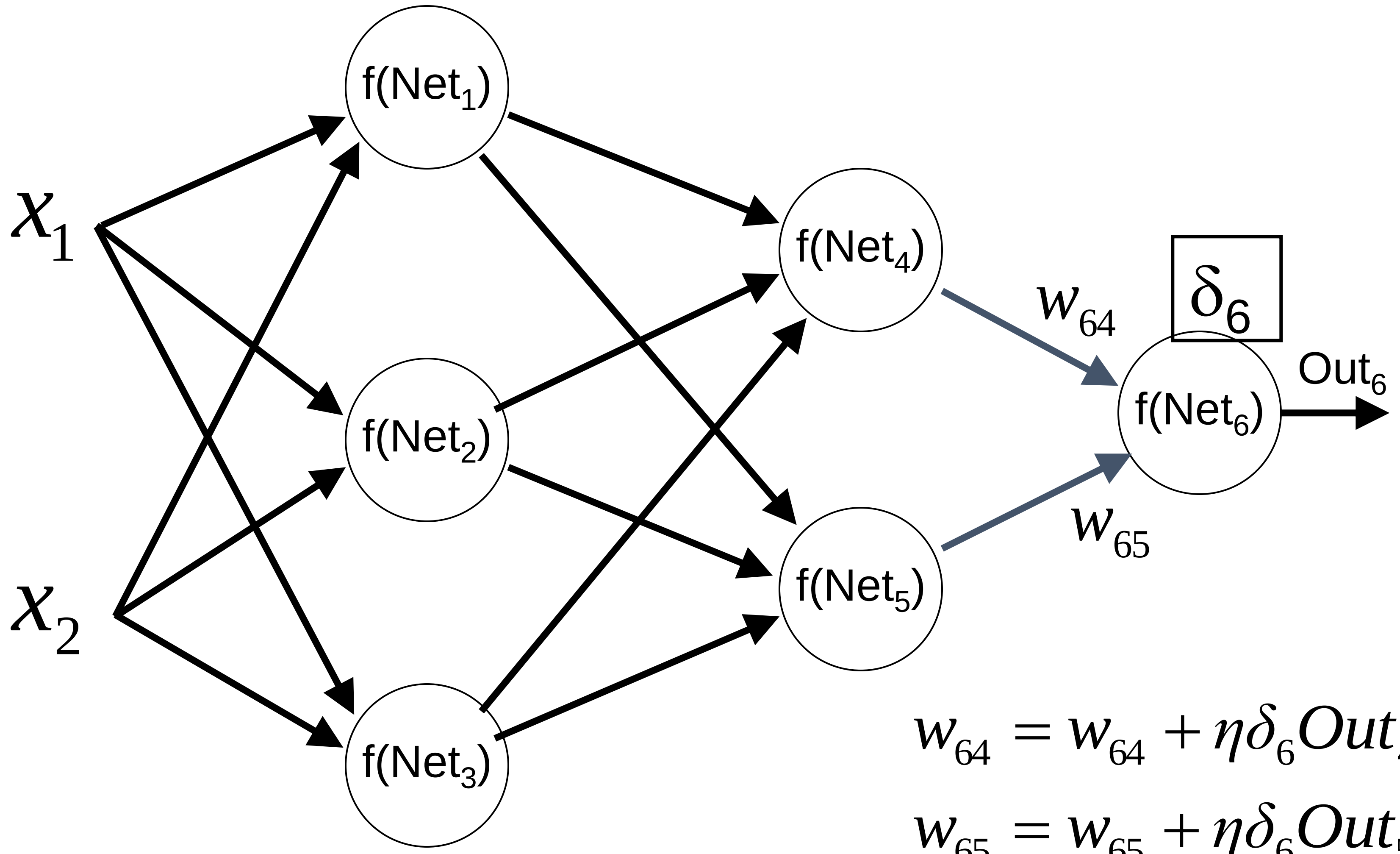


$$w_{51} = w_{51} + \eta\delta_5 Out_1$$

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

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

BP illustration – Weight update (6)



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

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

Advantages vs. 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)

When using ANNs?

- 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