Machine Learning and Data Mining
(IT4242E)

Quang Nhat NGUYEN

quang.nguyennhat@hust.edu.vn

Hanoi University of Science and Technology

School of Information and Communication Technology

Academic year 2021-2022

The course's content:

- Introduction
- Performance evaluation of the ML/DM system
- Regression problem
- **Classification problem**
  - **Artificial neural network**
- Clustering problem
- Association rule mining problem

Artificial neural network – Introduction (1)

- Artificial neural network – ANN
  - Simulation of biological neuron systems (the human brains)
  - ANN is a structure/network made up of a number of neurons linked together
- A neuron:
  - Has an input/output characteristics
  - Performs a local calculation (i.e., function)
- The output value of a neuron is determined by:
  - Its input/output characteristics
  - Its links to other neurons
  - (Possibly) additional inputs

Artificial neural network – Introduction (2)

- ANN can be viewed as a highly distributed and parallel information processing structure
- ANN is able to learn, recall, and generalize from training data – by assigning and adjusting (i.e., adapting) the weight values (i.e., importance degrees) of the links between neurons
- The function of an ANN is determined by:
  - Topology of the neural network
  - Input/output characteristics of each neuron
  - Learning (i.e., training) strategy
  - Training data

ANN – Typical applications (1)

- Image processing and Computer vision
  - Examples: Image segmentation and analysis, object detection, object recognition, human activity recognition and prediction, character recognition, face recognition, etc.
- Natural language understanding
  - Examples: Text classification, Name entity recognition (NER), Sentiment analysis, Question answering, etc.
- Speech understanding
  - Examples: Speech recognition, etc.
- Signal processing
  - Example: Signal analysis and seismic morphology for earthquake prediction
- Pattern recognition
  - Examples: Feature extraction, Radar signal analysis and classification, Fingerprint recognition, etc.

ANN – Typical applications (2)

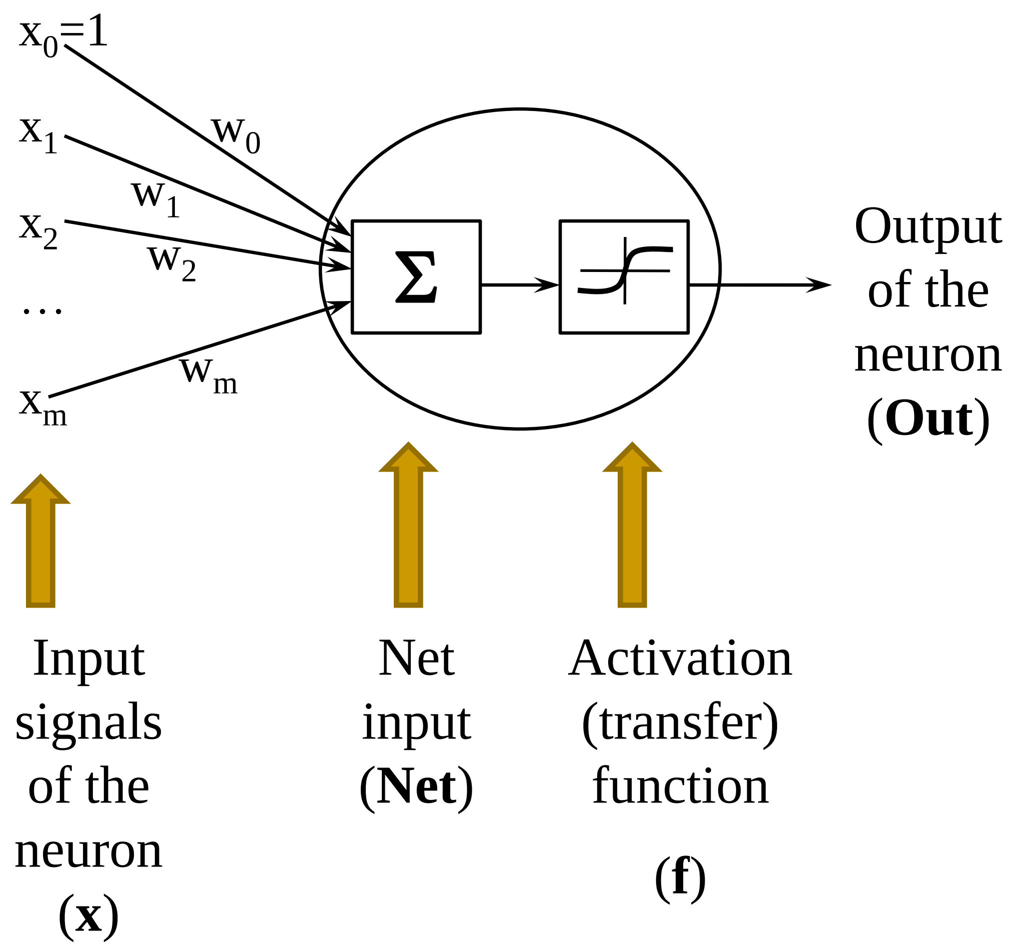
- Financial systems
  - Examples: Stock market analysis, real estate evaluation, credit card access control, stock trading, etc.
- Medical systems
  - Examples: ECG signals analysis and understanding, diseases diagnostics, medical images processing, etc.
- Energy systems
  - Examples: System status evaluation, problem detection and troubleshooting, workload prediction, safety evaluation, etc.
- Military systems
  - Examples: Detection of mines, radar noise classification, etc.
- ...(*and many other application areas*!)

DNN – Break-through application domains

- Deep neural networks (DNN) achieves break-through results in some application domains, such as:
  - Computer vision
  - Natural language understanding
  - Speed recognition

Structure and operation of a neuron

- **Input signals** of a neuron (x_i, i=1..m)
  - Each input signal x_i associates with a weight w_i
- **Adjustment (bias) weight** w_0 (for x_0=1)
- **Net input** is an integrated function of the input signals – `Net(`**`w,x`**`)`
- **Activation (transfer) function** calculates the output value of the neuron – `f(Net(`**`w,x`**`))`
- **Output** of the neuron: `Out=f(Net(`**`w,x`**`))`

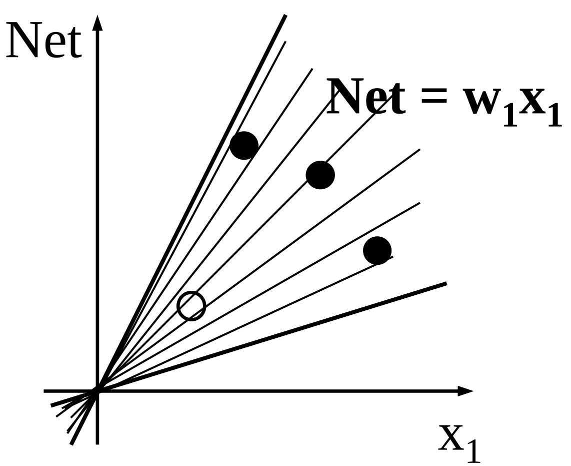


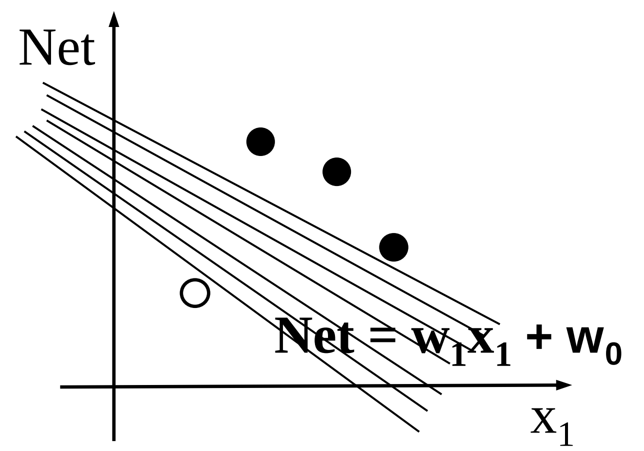
Net input and Bias

- The net input is usually calculated by a linear function:

$$Net = w_0 + w_1x_1 + w_2x_2 + \ldots + w_mx_m = w_0.1 + \sum_{i=1}^{m} w_ix_i = \sum_{i=0}^{m} w_ix_i$$

- Meaning of the bias
  - → Family of separation functions $\texttt{Net=w}_1\texttt{x}_1$ *cannot separate* examples into 2 classes
  - → But: Family of functions $\texttt{Net=w}_1\texttt{x}_1\texttt{+w}_0$ can do that!



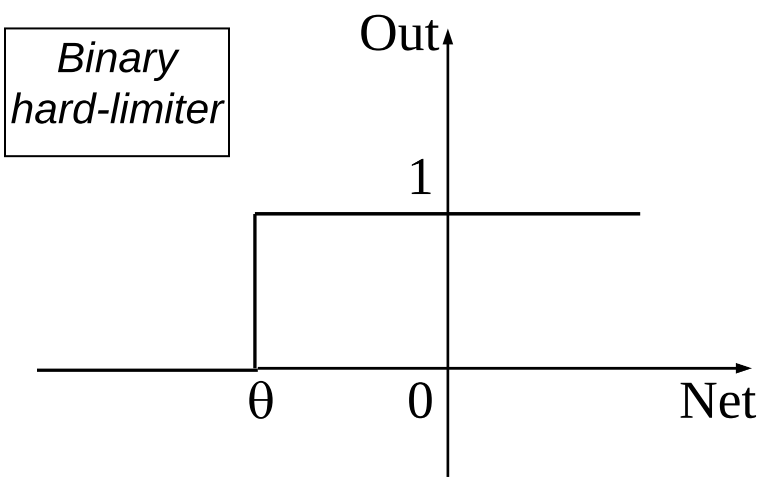


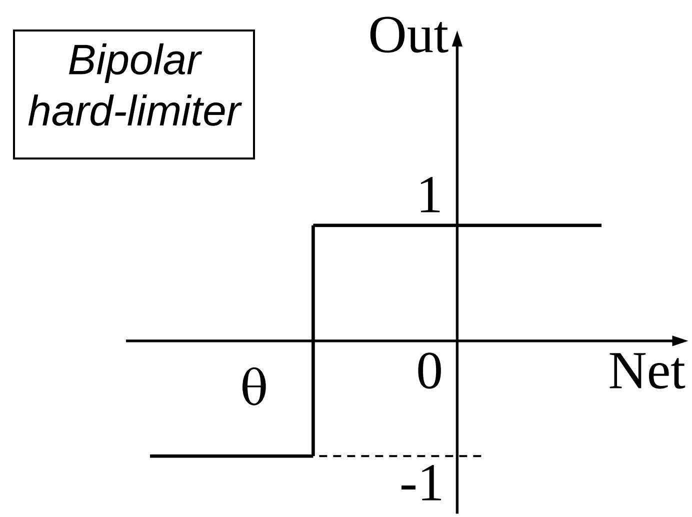
Activation function: Hard-limiter

- Also called Threshold function
- The output value takes either of the 2 values
- θ is a threshold value
- **Disadvantage**: Discontinuous, and its derivative is discontinuous

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

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

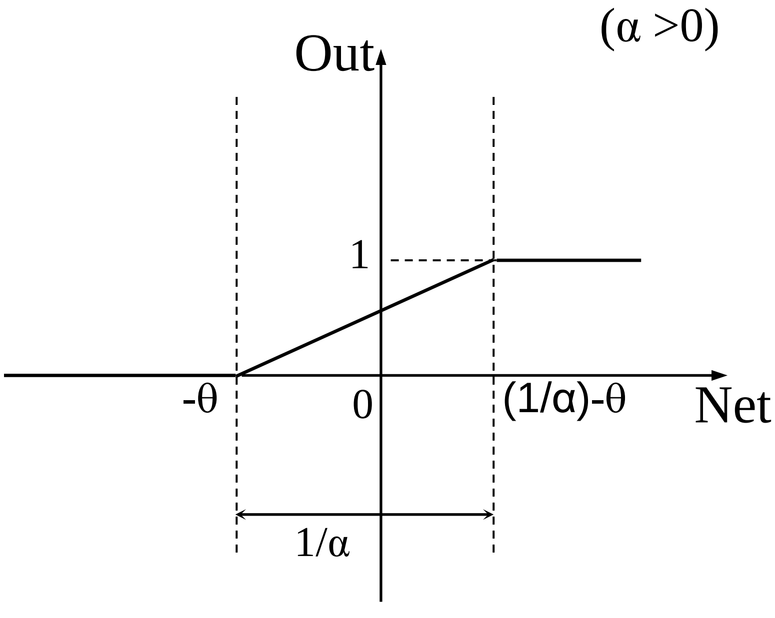




Activation function: Threshold logic

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

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

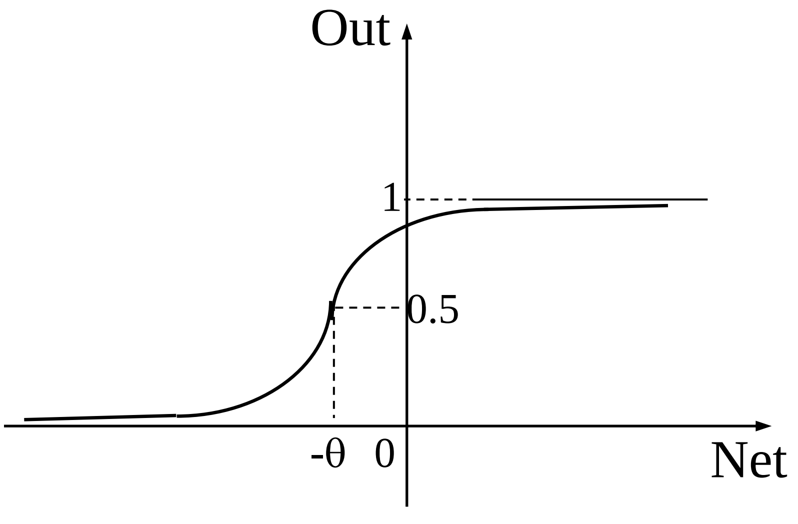


- Also called Saturating linear function
- A combination of 2 activation functions: Linear and Hard-limiter
- α defines the slope of the linear range
- **Disadvantage**: Continuous, but its derivative is discontinuous

Activation function: Sigmoidal (Logistic)

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

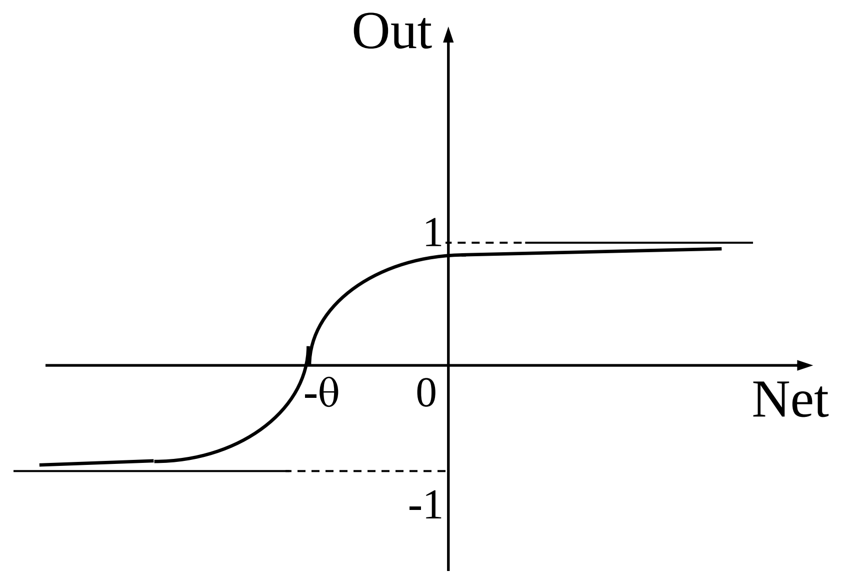
- Very popularly used
- Parameter α defines the slope
- The output value is in range of (0,1)
- **Advantages:**
  - Continuous, and its derivative is continuous
  - The derivative of a sigmoidal function can be represented by a function of itself



Activation function: Hyperbolic tangent

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

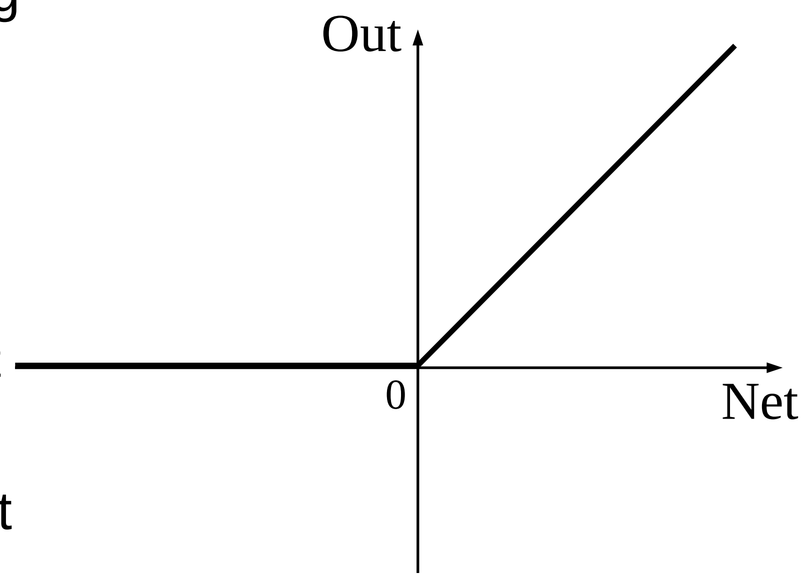
- Also popularly used
- Parameter α defines the slope
- The output value is in range of (-1,1)
- **Advantages:**
  - Continuous, and its derivative is continuous
  - The derivative of a `tanh` function can be represented by a function of itself



Activation function: Rectified linear unit (ReLU)

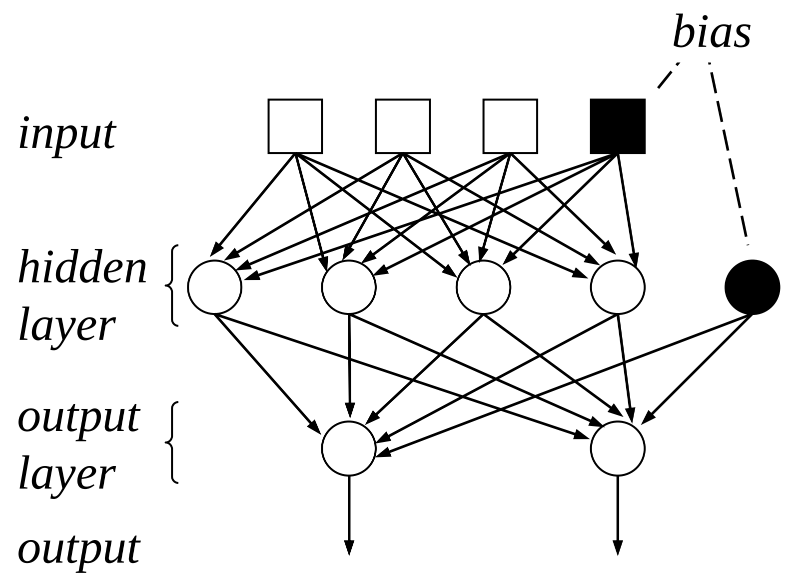
Out(Net) = relu(Net) = max(0, Net)

- Very popularly used in deep learning
- **Advantages (compared to sigmoid, tanh):**
  - Avoid the problem "*Vanishing gradients*"
  - Reduce much of computational cost
- Continuous, but its derivative is discontinuous (i.e., for negative input values)
  - Assumption: Derivative is 0 for negative input values



ANN – Topology (1)

- Topology of an ANN is defined by:
  - Number of input and output signals
  - Number of layers
  - Number of neuron for each layer
  - Number of weights of the links for each neuron
  - How neurons (in a layer, or between layers) are connected
  - Which neurons receive error correction signals
- An ANN has:
  - 1 input layer
  - 1 output layer
  - Zero, one, ore more hidden layer(s)



Example: An ANN having 1 hidden layer

- Input: 3 signals
- Output: 2 values
- In total, this ANN has 6 neurons:
  - 4 in the hidden layer
  - 2 in the output layer

ANN – Topology (2)

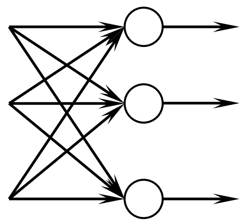
- A layer contains a group of neurons
- A hidden layer is such layer that is a layer located between the input layer and the output layer
- The hidden nodes do not interact directly with the external environment (of the neural network)
- An ANN is called ***fully connected*** if all outputs from one layer are connected to every neuron of the next layer

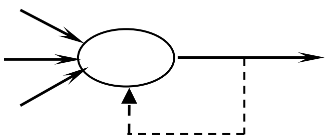
ANN – Topology (3)

- An ANN is called a ***feed-forward*** network if there is not any output of a neuron that is the input of another neuron in the same layer (or a previous layer)
- When the output of a node links backward as the input of another node in the same layer (or a previous layer), then it is a ***feedback*** network
  - If the feedback is an input to the nodes of the same layer, then it is ***lateral feedback***
- Feedback networks have closed loops called ***recurrent networks***

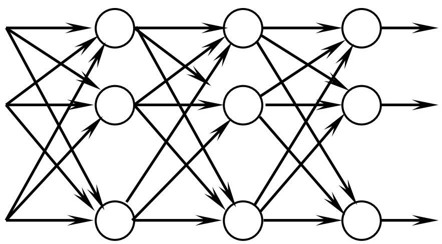
Network topology – Examples

2-layer feedforward network

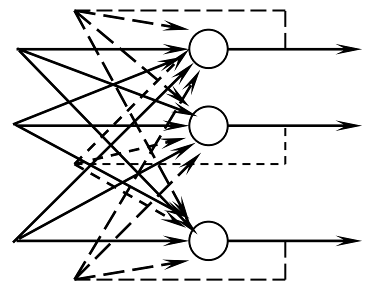




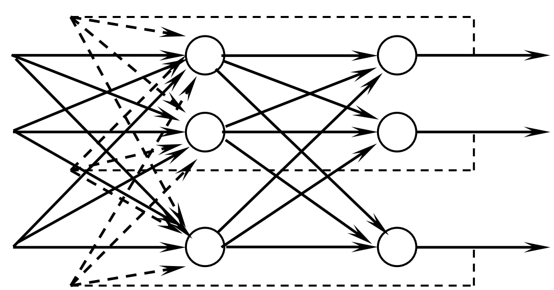
A neuron with feedback to itself



multi-layer feedforward network



2-layer recurrent network



Multi-layer recurrent network

ANN – Learning rules

- 2 learning types for ANNs
  - *Parameter learning*
    - → The goal is to learn (i.e., to adapt) the weights of links in the neural network
  - *Structure learning*
    - → The goal is to adapt the network structure, including the number of neurons and the types of connections between them
- These two types of learning can be done simultaneously or separately
- Most of the learning rules for ANN belong to parameter learning
- In this lecture, we consider just parameter learning

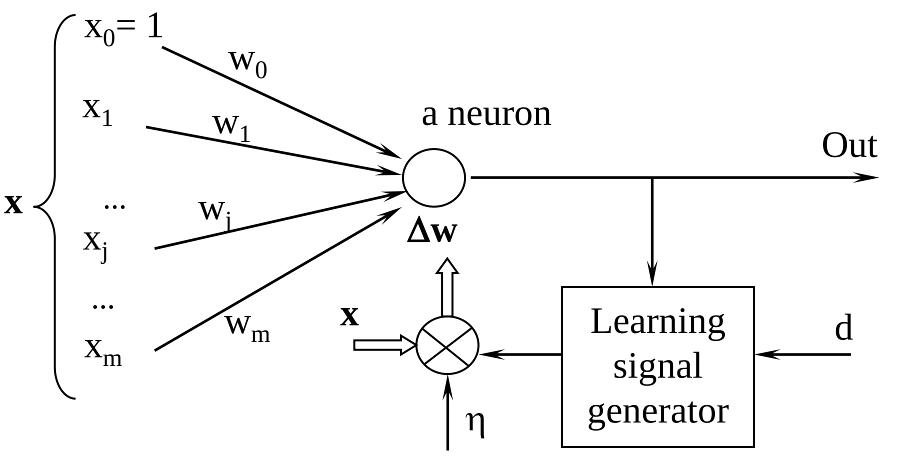
General weight learning rule

- At learning step (t), the adjustment of the weights vector ***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 learning rate**



- The learning signal r is a function of ***w***, ***x***, and the expected output d

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

- General weight learning rule:

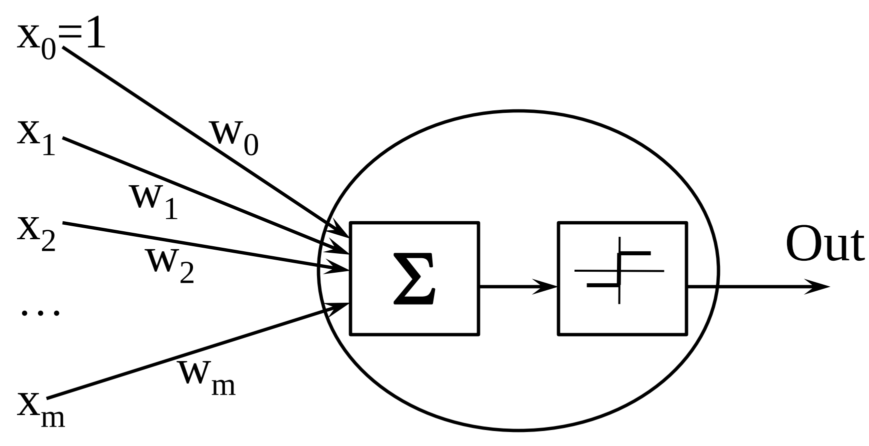
$$\Delta\mathbf{w}^{(t)} = \eta.g(\mathbf{w}^{(t)},\mathbf{x}^{(t)},d^{(t)}).\mathbf{x}^{(t)}$$

Note: x_j can be:

- An input signal, or
- An output value of another neuron

Perceptron

- A perceptron is the simplest kind of ANNs (i.e., containing only 1 neuron)

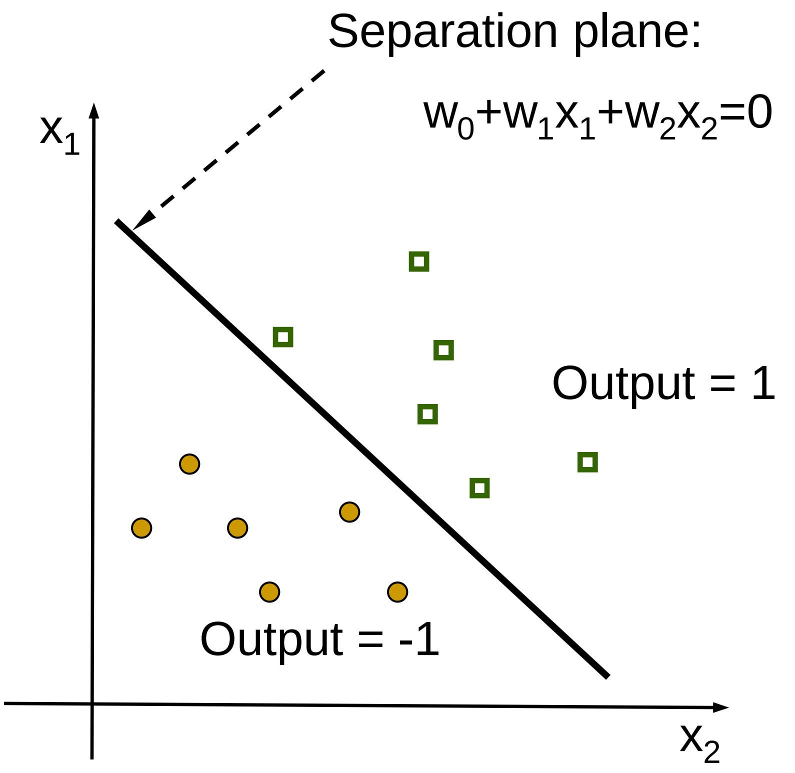


- Use the hard-limiter activation function

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

- For an example ***x***, the output value of perceptron:
  - 1, if Net(**w**,**x**)>0
  - -1, if otherwise

Perceptron – Illustration



Perceptron – Learning algorithm

- Given a training set D= {(**x**,d)}
  - ***x*** is an input vector
  - *d* is the expected output value (-1 or 1)
- The learning process of a perceptron aims to determine a weights vector that allows the perceptron produces the exact output value (-1 or 1) for each training example
- For a training example ***x*** that is **correctly** classified by the perceptron, the weights vector ***w*** does not change
- If d=1 but the perceptron outputs -1 (Out=-1), then ***w*** needs to change so that Net(**w**,**x**) increases
- If d=-1 but the perceptron outputs 1 (Out=1), then ***w*** needs to change so that Net(**w**,**x**) decreases

Perceptron_incremental(D, η)

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

do

$\Delta\mathbf{w} \leftarrow 0$

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

Compute the real output value Out

if ($Out \neq d$)

$\Delta\mathbf{w} \leftarrow \Delta\mathbf{w} + \eta(d - Out)\mathbf{x}$

end for

$\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w}$

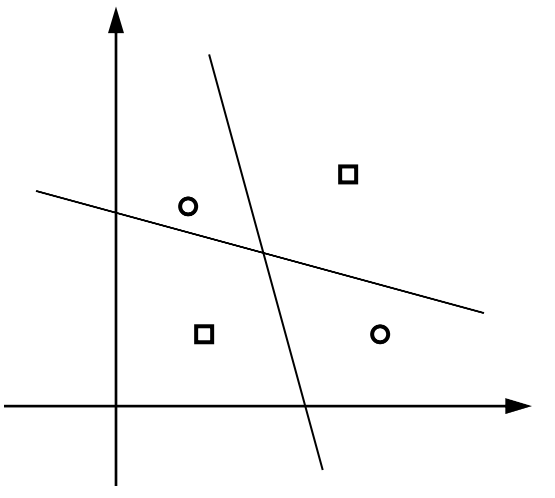
until all the training instances in D are correctly classified

return $\mathbf{w}$

Perceptron – Limitation

- The learning algorithm of perceptron is proved to converge if:
  - The training examples are linearly separable,
  - Using a small-enough learning rate η
- The learning algorithm of perceptron may not converge if the training examples are not linearly separable
- Then, use **the delta rule**
  - Guarantee to converge to an approximation of the target function
  - The delta rule uses the **gradient descent** strategy to find in the space of hypotheses (i.e., weight vectors) a weight vector most appropriate to the training examples

A perceptron cannot classify correctly this training set!



Error/loss function

- Let's consider an ANN that has *n* output neurons
- For a training example (**x**,d), the **training error** caused by the (current) 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** caused by the (current) weights vector **w** for 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 that has:
  - The direction of going up of the slope
  - The length is proportional to the slope
- Gradient ∇E determines the direction that causes the **steepest increase** of the error 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 weights (links) of the network

- Therefore, the direction that causes the **steepest decrease** is the negation of the gradient of E

$$\Delta \mathbf{w} = -\eta \,.\, \nabla E(\mathbf{w});$$

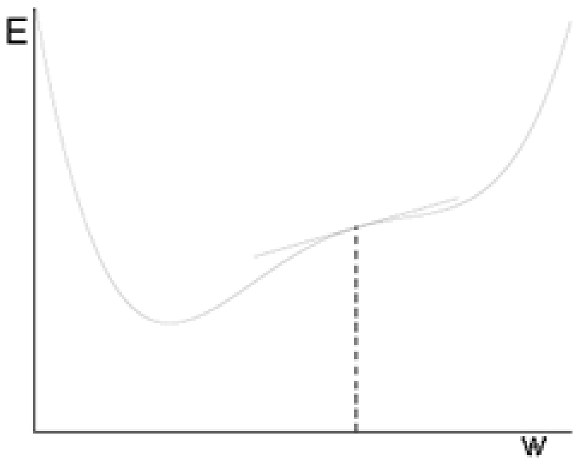
$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}, \quad \forall i = 1..N$$

- <u>Condition</u>: The activation functions used in the network must be continuous to the weights, and their derivatives are also continuous

Gradient descent – Illustration

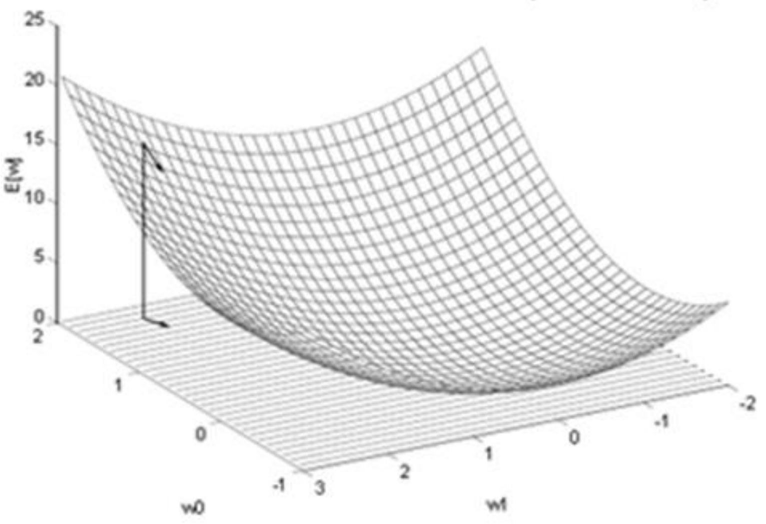
1-dimensional space

E(w)



2-dimensional space

$E(w_1,w_2)$



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: Number of epochs, Error threshold, ...

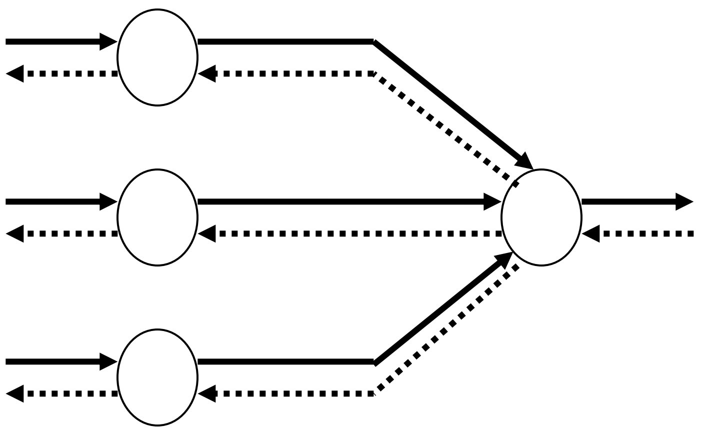
Multi-layer ANN and Back-propagation algorithm

- A perceptron can represent only a linear separation function
- A multi-layer ANN learns by the back-propagation (BP) algorithm can represent a highly non-linear separation function
- The BP learning algorithm is used to learn the weights of a multi-layer ANN
  - *The network topology is fixed* (i.e., the neurons and their links are fixed)
  - For each neuron, *the activation function must have its derivative continuous*
- The BP algorithm applies the *gradient descent* strategy for the weights update rule
  - To minimize the error (difference) between the real outputs and the expected ones for the training examples

Back-propagation learning algorithm (1)

- The back-propagation learning algorithm searches for a weights vector that **minimizes the error** made by the system for the training set
- The BP algorithm consists of 2 phases:
  - **Signal forward propagation**. The input signals (i.e., a vector of input values) <u>are propagated forward</u> from the input layer to the output one (passing through the hidden layers)
  - **Error backward propagation**
    - Based on the expected output value of the input vector, the system computes the error
    - Starting from the output layer, the error <u>is propagated backward</u> through the network, from a layer to another previous one, until the input layer
    - This error back-propagation is done by computing (recursively) the local gradient of each neuron

Back-propagation learning algorithm (2)



Signal forward propagation:

- Propagate forward the input signals through the network

Error backward propagation:

- Compute the error at the output layer
- Propagate backward the error through the network

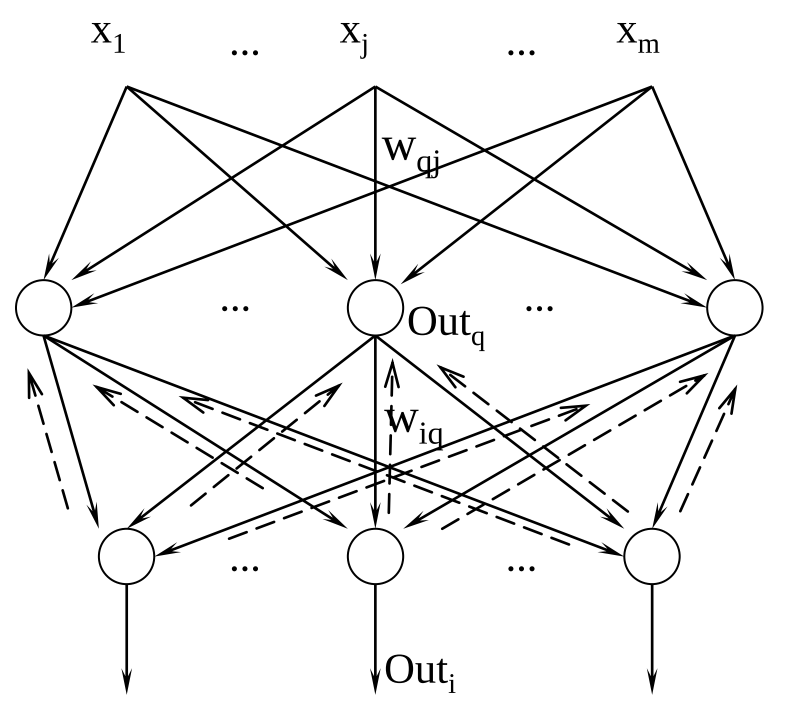
BP algorithm – Network topology

- Let's consider a 3-layer ANN for illustration of the BP alg.
- *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 link from the input signal x_j to the hidden neuron z_q
- w_{iq} is the weight of the link from the hidden neuron z_q to the output neuron y_i
- Out_q is the (local) output value of the hidden neuron z_q
- Out_i is the output of the network 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)



BP algorithm – Forward propagation (1)

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

- For an input vector ***x***, a hidden neuron z_q receives the net input:

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

…and produces a (local) output value:

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

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

BP algorithm – Forward propagation (2)

- The net input of the output neuron y_i:

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

- The neuron y_i produces the output value (i.e., which is an output value 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 the output values Out_i (i=1..n) are the actual output values of the network for the input vector $\boldsymbol{x}$

BP algorithm – Error computation

- For a training example ***x***
  - The error signals, caused by the difference between the expected output vector ***d*** and the actual output vector ***Out***, are computed
  - These error signals are *back-propagated* from the output layer to the previous ones to update the weights (of the links)

- To present the error signals and their back propagation, we need to define an error (loss) 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 propagation (1)

- For the gradient descent method, the weights of the links **from the last hidden layer to the output one** 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: The sign "–" is already integrated in the value of $\partial E/\partial Out_i$)

- $\boldsymbol{\delta_i}$ is the **error signal** of the neuron y_i <u>at the output layer</u>

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

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

BP algorithm – Backward propagation (2)

- To update the weights of the links **from the input layer to a hidden one (or from a hidden layer to a next hidden one)**, we also apply the 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 formula of the error function *E(**w**)*, we see that each error component (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 propagation (3)

- By applying 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 the neuron z_q at a 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 of the neuron z_q at a hidden layer, and $f'(\mathrm{Net}_q) = \partial f(\mathrm{Net}_q) / \partial \mathrm{Net}_q$

BP algorithm – Backward propagation (4)

- In the formulae of computation of the error signals δ_i and δ_q, *the error signal of a neuron at a hidden layer* **is different from** *the error signal of a neuron at the output layer*
- Because of this difference, the weights update procedure of the BP algorithm is called *the general delta learning rule*
- The error signal δ_q of the neuron z_q at a hidden layer is defined by:
  - The error signals δ_i of the neurons y_i at the output layer (that the neuron z_q links to), and
  - The weights w_{iq}
- An important characteristic of the BP algorithm: **The weights update rule is local**
  - To compute the change (update) of the weight of a link, the system needs to use just the values at the 2 ends of that link!

BP algorithm – Backward propagation (5)

- The process of computing the error signals mentioned above can be extended (generalized) easily for an ANN that has more than 1 hidden layer

- The general form of the weights update rule of the BP algorithm is:

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

  - *b* and *a* are the 2 indexes corresponding to the 2 ends of the link (*b*→*a*) (from neuron (or input signal) *b* to neuron *a*)
  - x_b is the output value of a neuron at a hidden layer (or an input signal) *b*,
  - δ_a is the error signal of neuron *a*

Back_propagation_incremental(D, η)

The ANN consists of Q layers, $q = 1,2,...,Q$

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

The ANN has m input signals and n output neurons

${}^{q}w_{ij}$ is the weight of the link from neuron j at layer $(q\text{-}1)$ to neuron i at layer q

Step 0 (Initialization)

Select the error threshold $E_{threshold}$ (i.e., the maximum acceptable error)

Initialize the the weights by small and random values

Assign E=0

Step 1 (Beginning of an epoch)

Apply the input vector of the training example k to the input layer (q=1)

${}^{q}\mathrm{Out}_\mathrm{i} = {}^{1}\mathrm{Out}_\mathrm{i} = \mathrm{x}_\mathrm{i}^{(k)}, \forall i$

Step 2 (Forward propagation of the input signals)

Forwardly propagate the input signals through the ANN, until receiving the output values of the ANN (at the output layer) ${}^{Q}Out_i$

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

Step 3 (Computation of the output error)

Compute the output error of the ANN and the error signal ${}^{Q}\delta_i$ of each neuron at 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 (Backward propagation of the output error)

Backwardly propagate the output error to update the weights and compute the error signals ${}^{q-1}\delta_i$ for the previous 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 (Check the termination condition at an epoch)

Check if the entire training set is exploited (i.e., if a training epoch is elapsed)

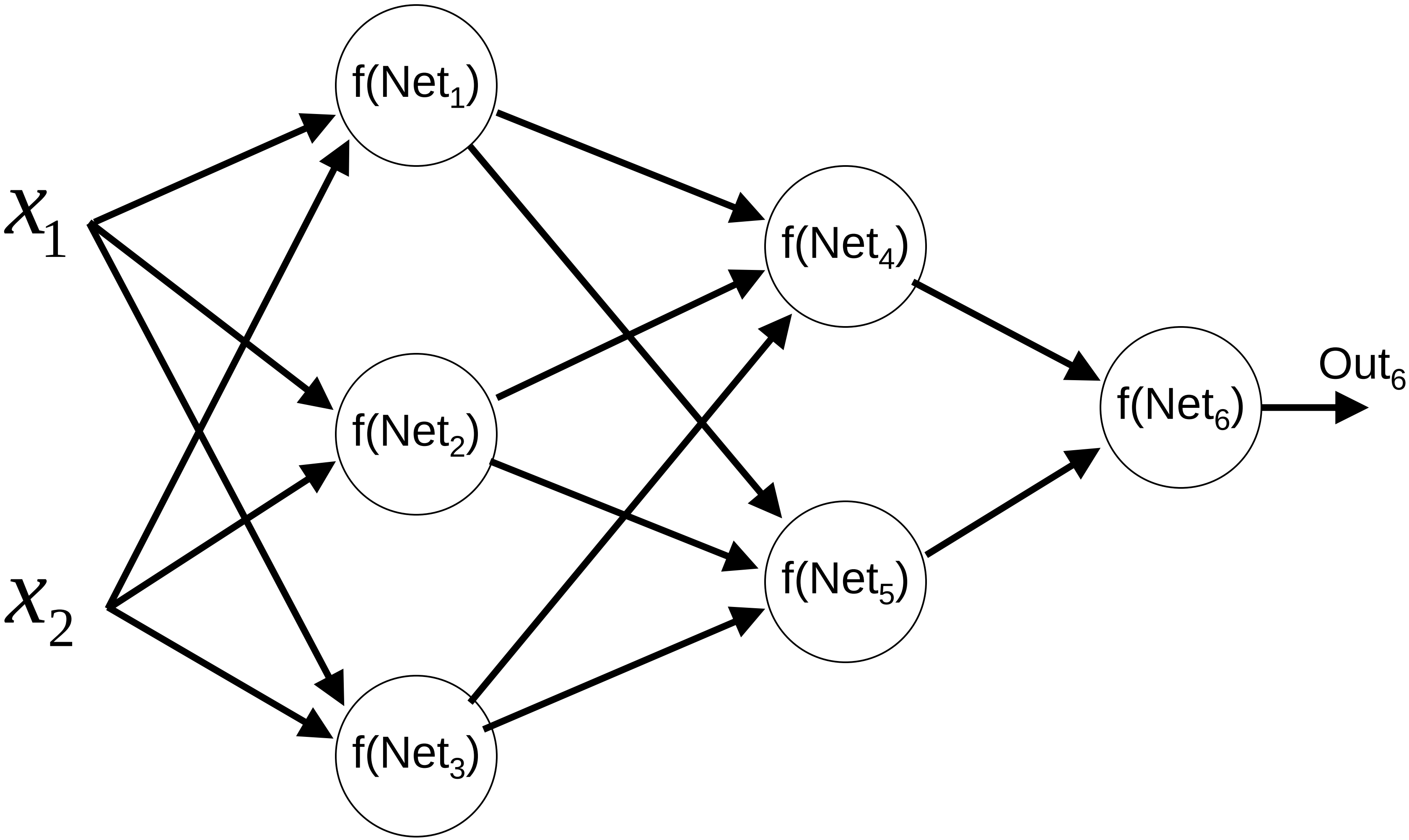
If the entire training set is exploited, then go to Step 6; otherwise, go to Step 1

Step 6 (Check the total error)

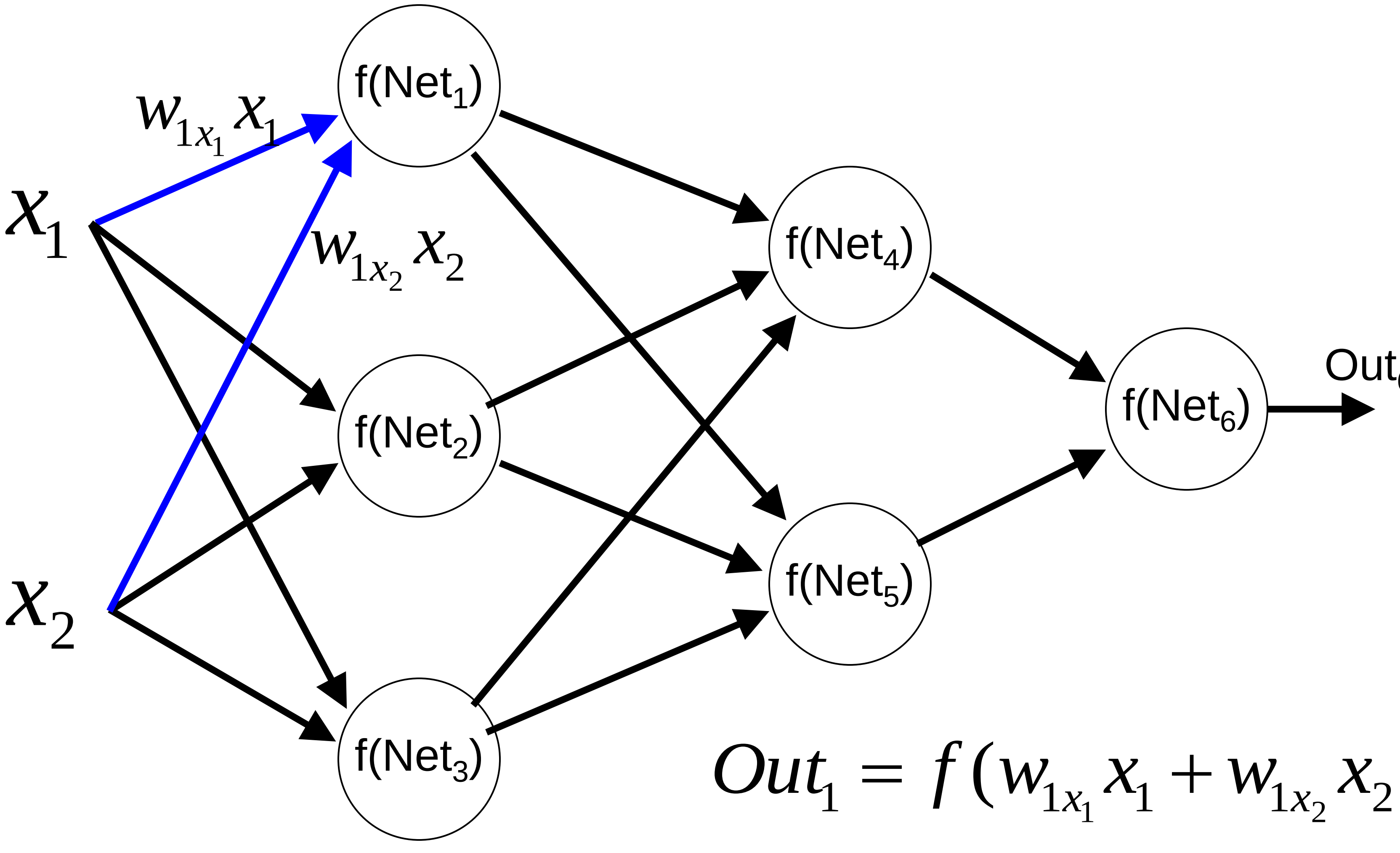
If the total error $E < E_{threshold}$, then the training process ends and returns the learned weights;

Otherwise, re-assign E=0, and start a new training epoch (i.e., go to Step 1)

BP algorithm – Forward propagation (1)

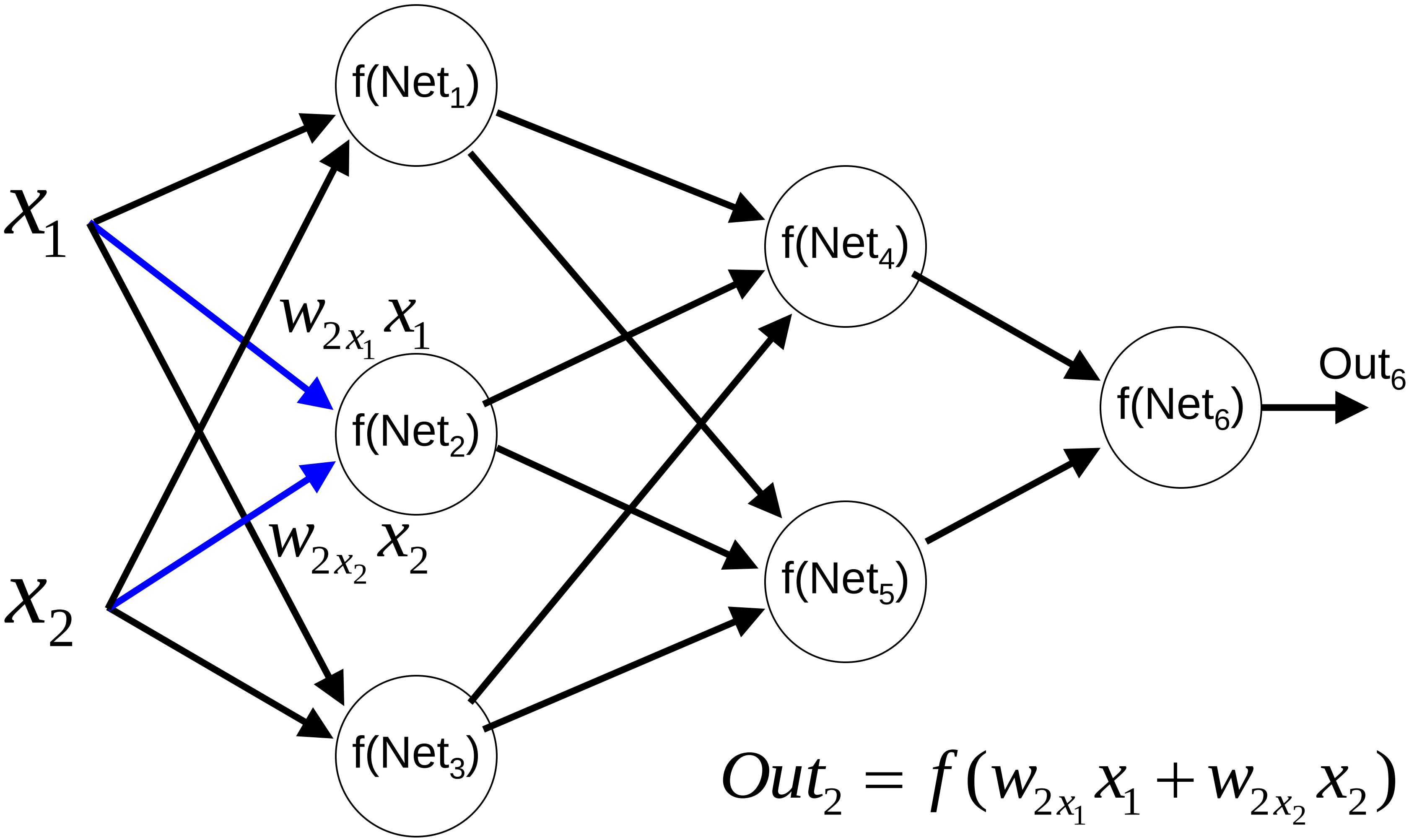


BP algorithm – Forward propagation (2)



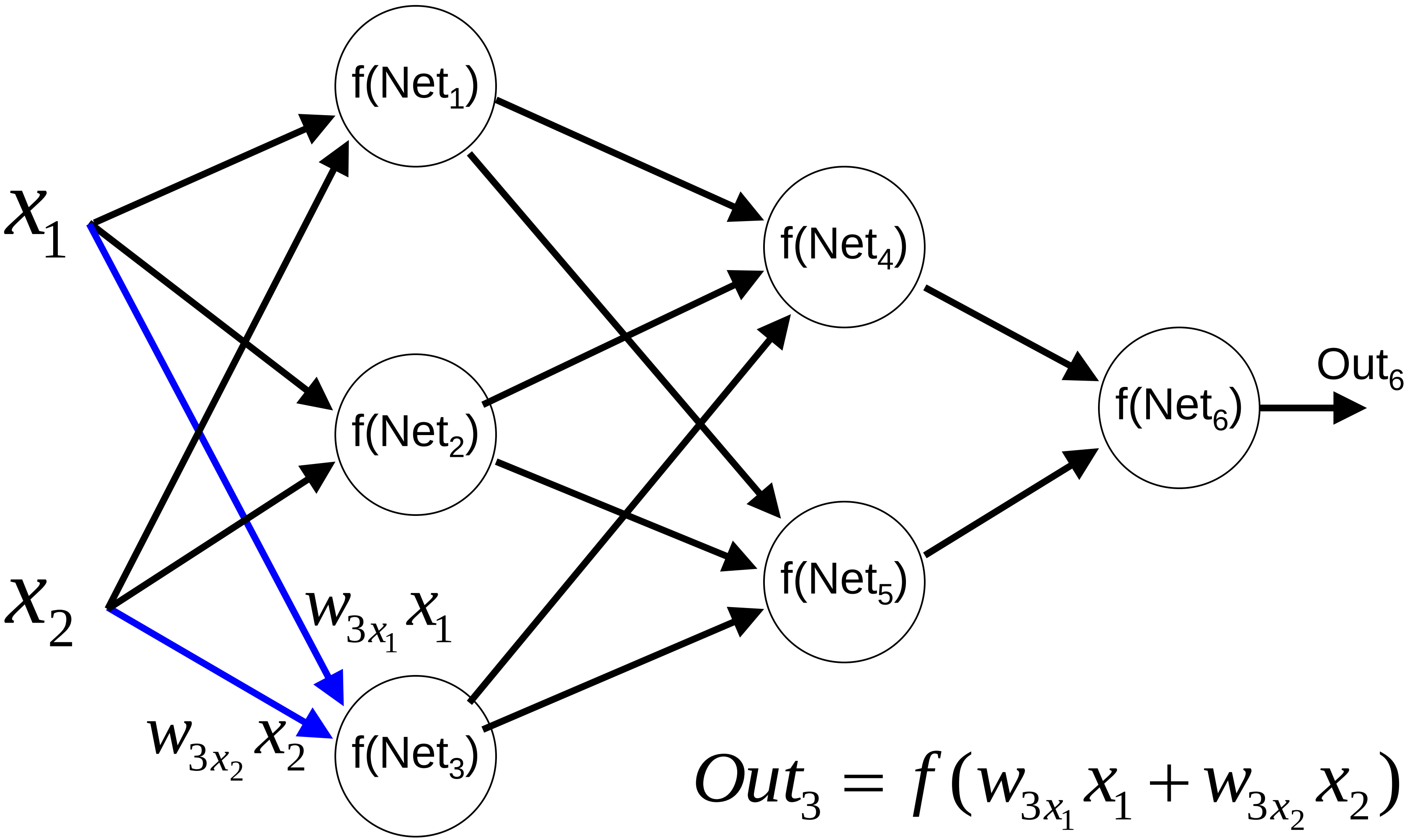
$$Out_1 = f(w_{1x_1} x_1 + w_{1x_2} x_2)$$

BP algorithm – Forward propagation (3)



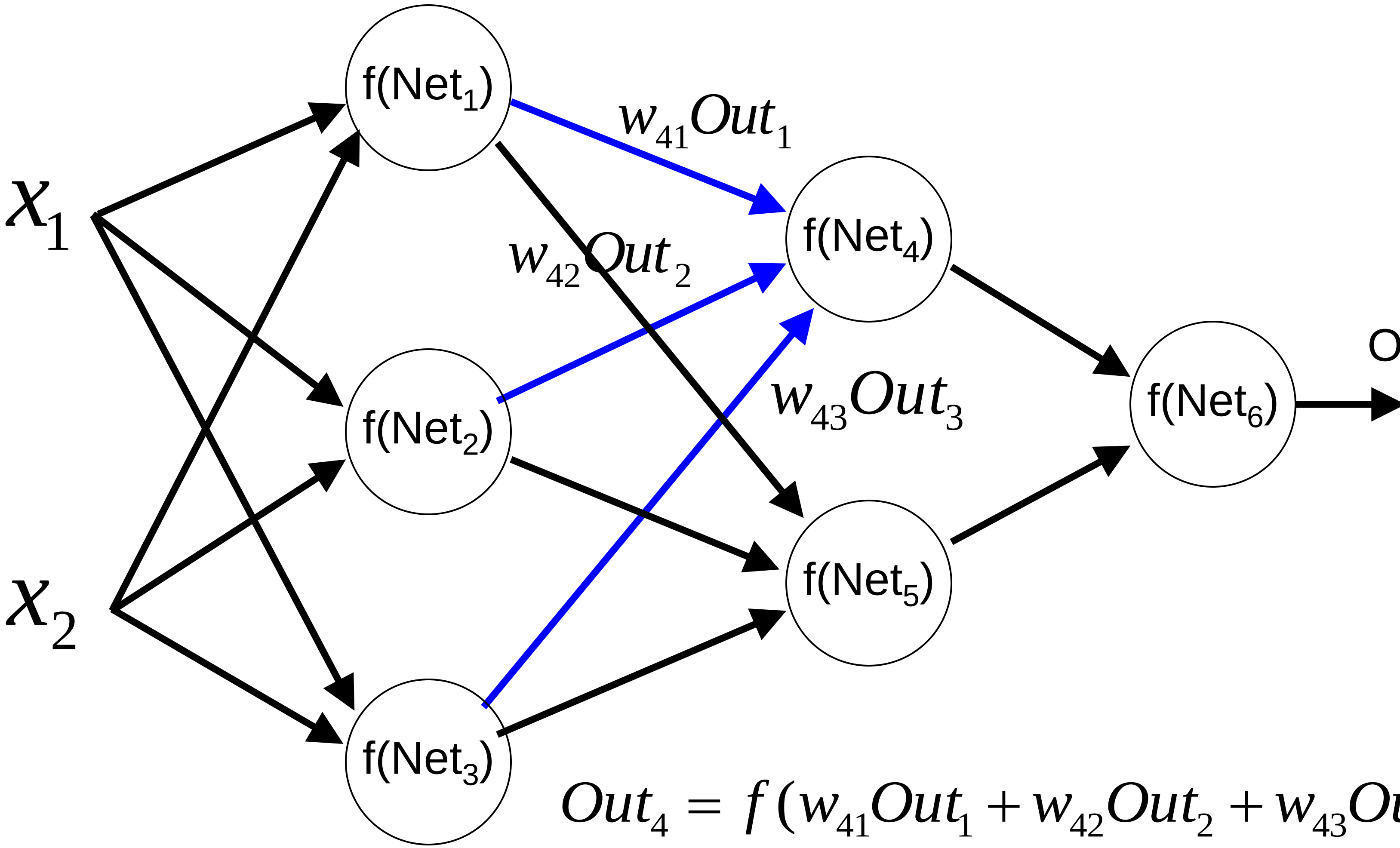
$$Out_2 = f\left(w_{2x_1}x_1 + w_{2x_2}x_2\right)$$

BP algorithm – Forward propagation (4)



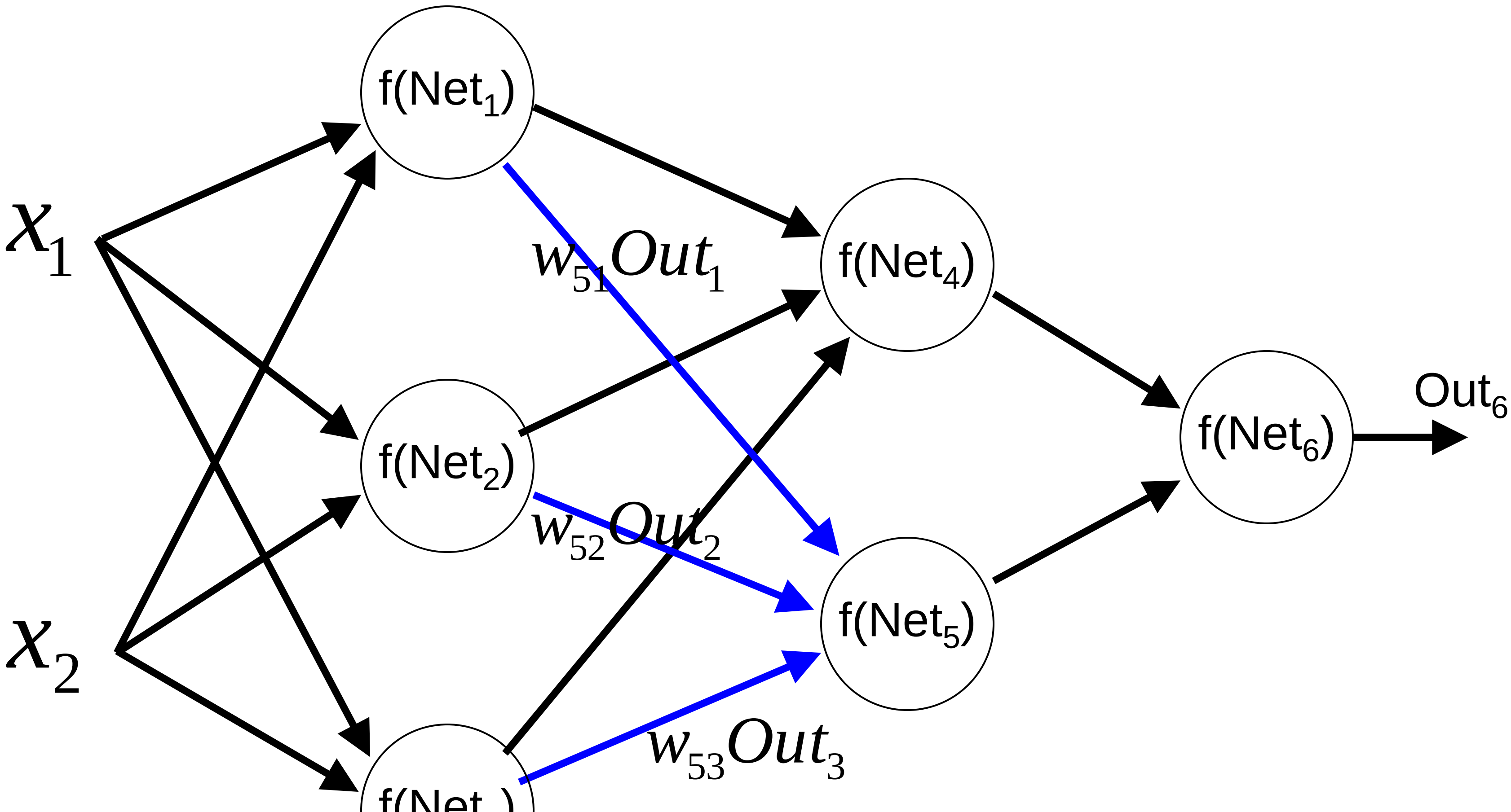
$$Out_3 = f(w_{3x_1}x_1 + w_{3x_2}x_2)$$

BP algorithm – Forward propagation (5)



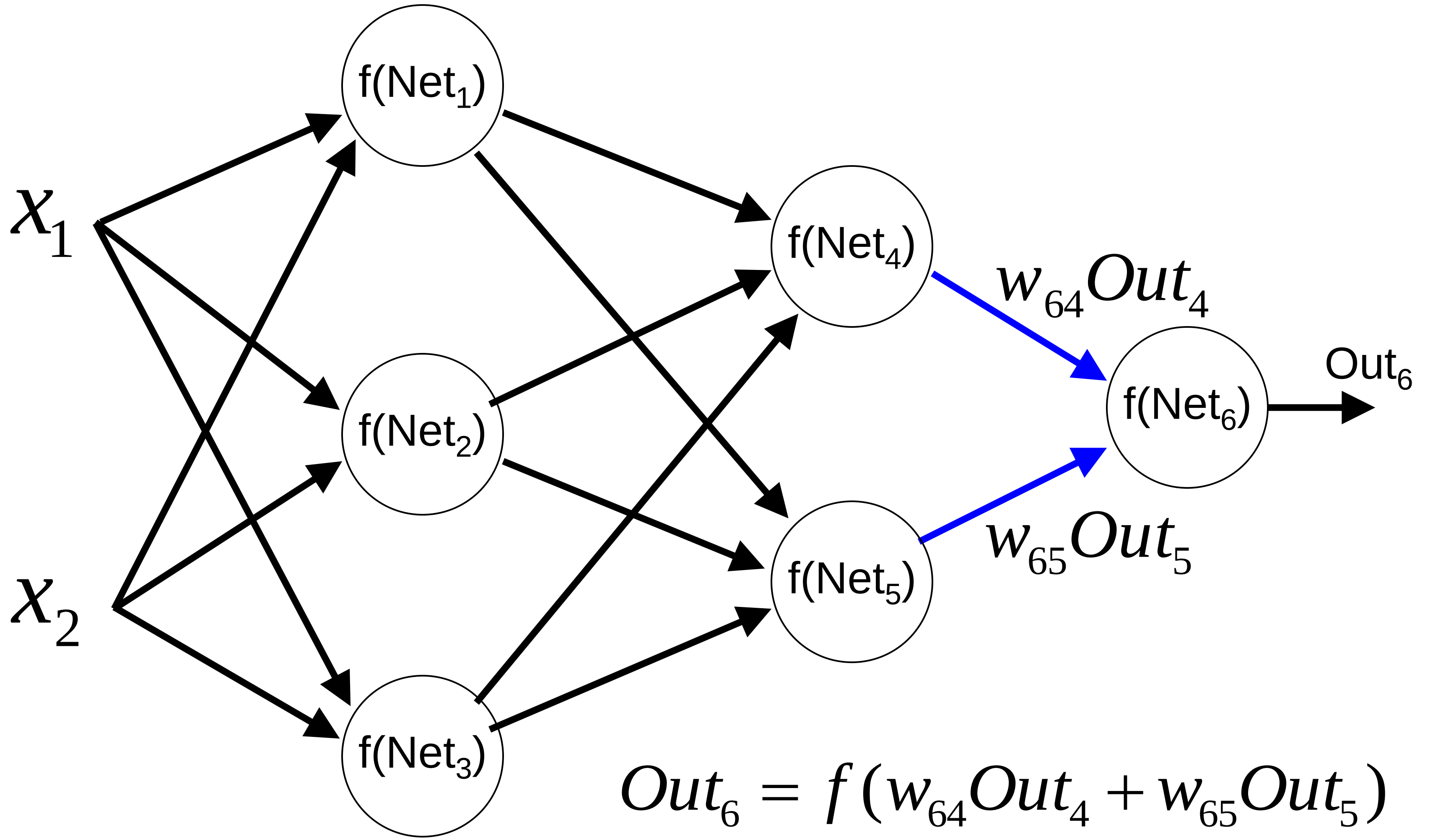
$$Out_4 = f(w_{41}Out_1 + w_{42}Out_2 + w_{43}Out_3)$$

BP algorithm – Forward propagation (6)



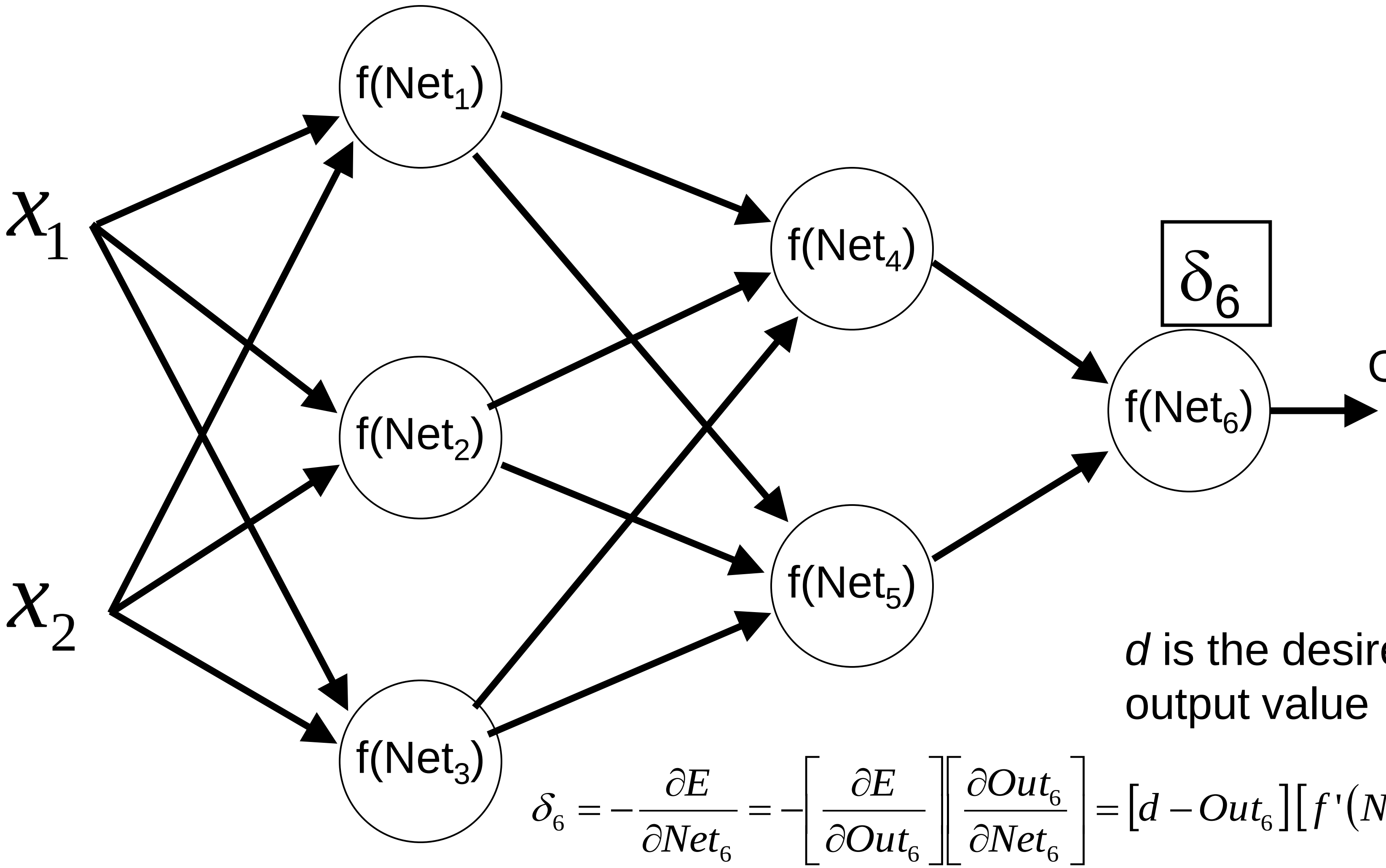
$$Out_5 = f\left(w_{51}Out_1 + w_{52}Out_2 + w_{53}Out_3\right)$$

BP algorithm – Forward propagation (7)



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

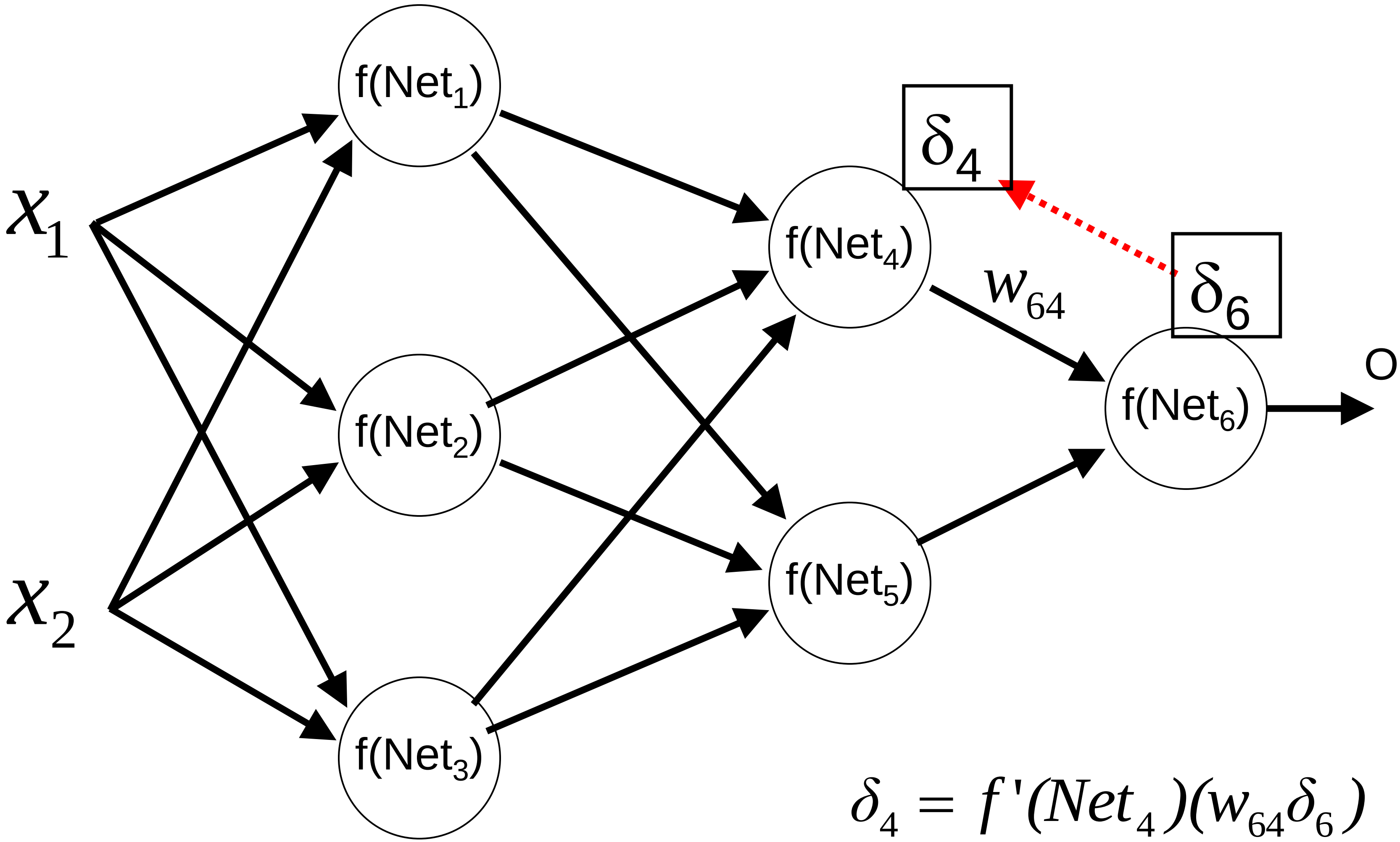
BP algorithm – Error computation



d is the desired output value

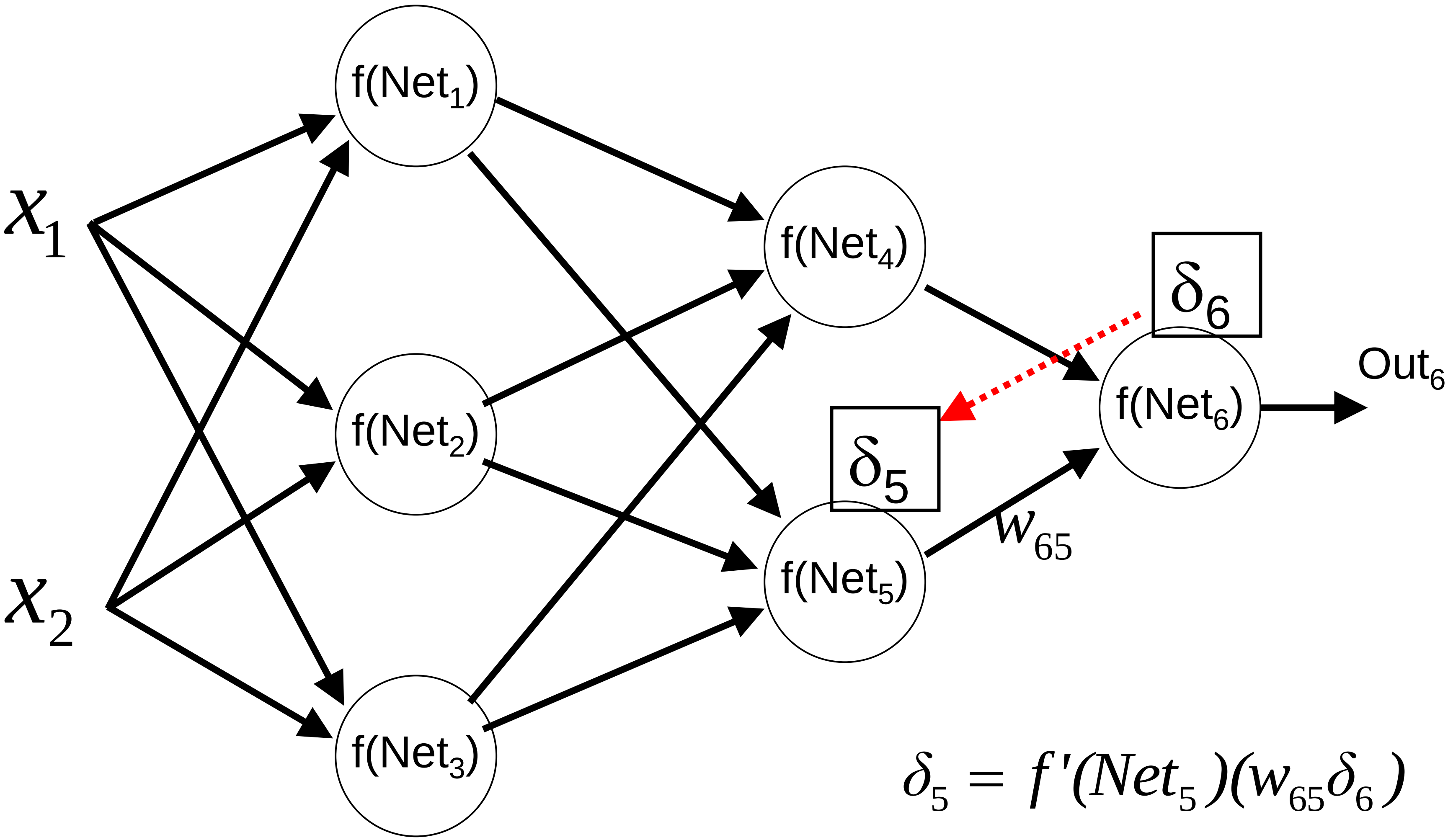
$$\delta_6 = -\frac{\partial E}{\partial Net_6} = -\left[\frac{\partial E}{\partial Out_6}\right]\left[\frac{\partial Out_6}{\partial Net_6}\right] = [d - Out_6][f'(Net_6)]$$

BP algorithm – Backward propagation (1)



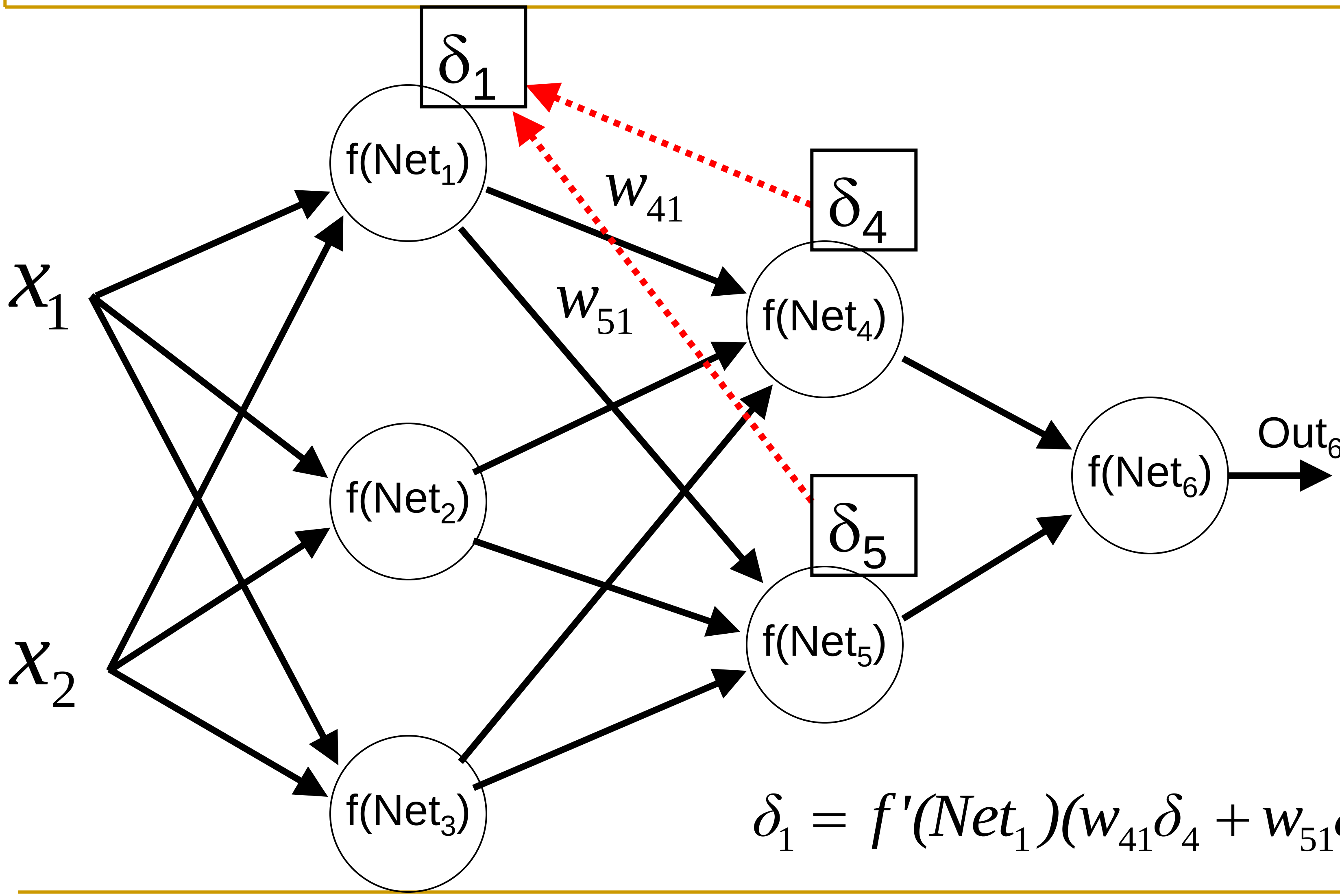
$$\delta_4 = f'(Net_4)(w_{64}\delta_6)$$

BP algorithm – Backward propagation (2)



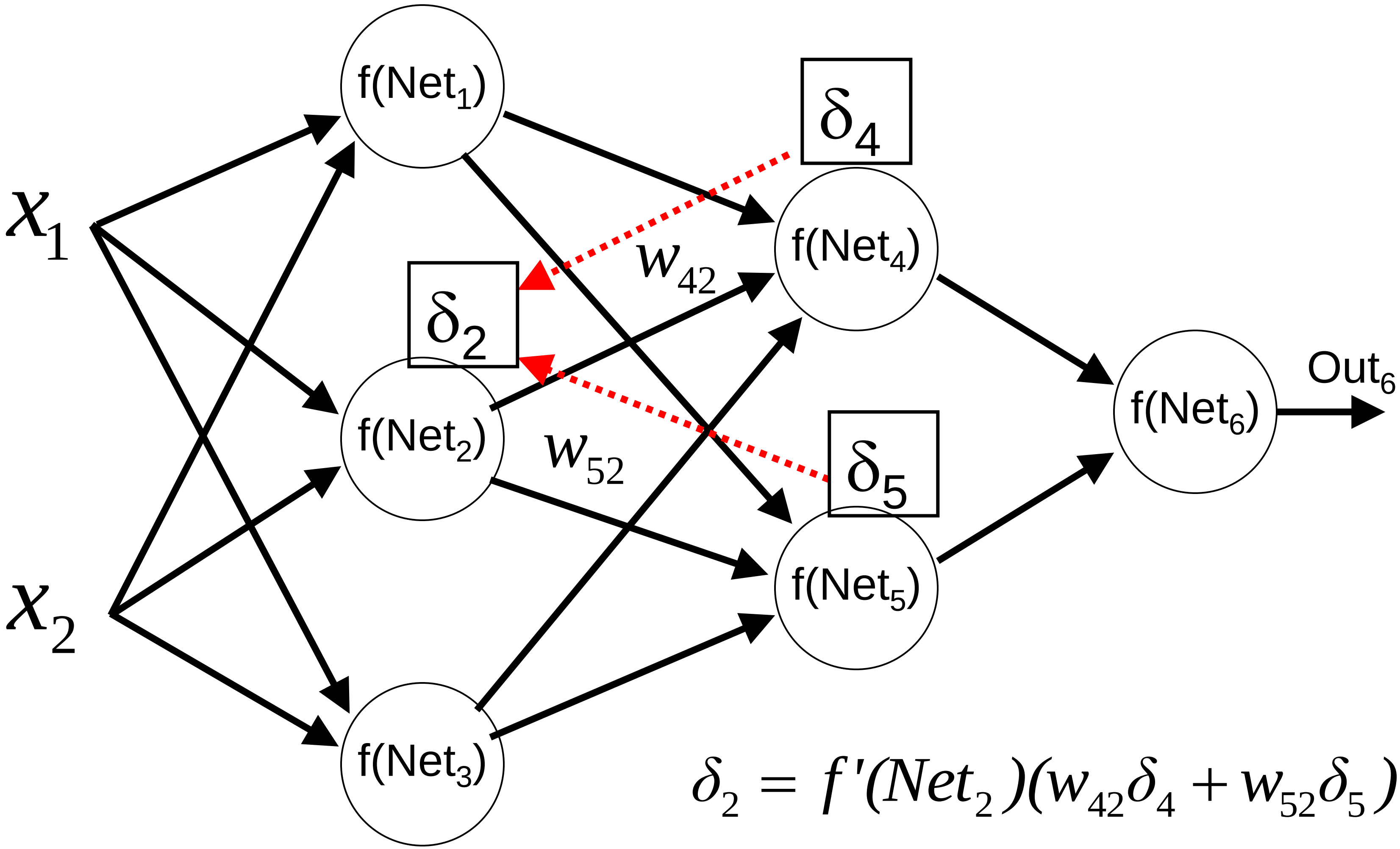
$$\delta_5 = f'(Net_5)(w_{65}\delta_6)$$

BP algorithm – Backward propagation (3)



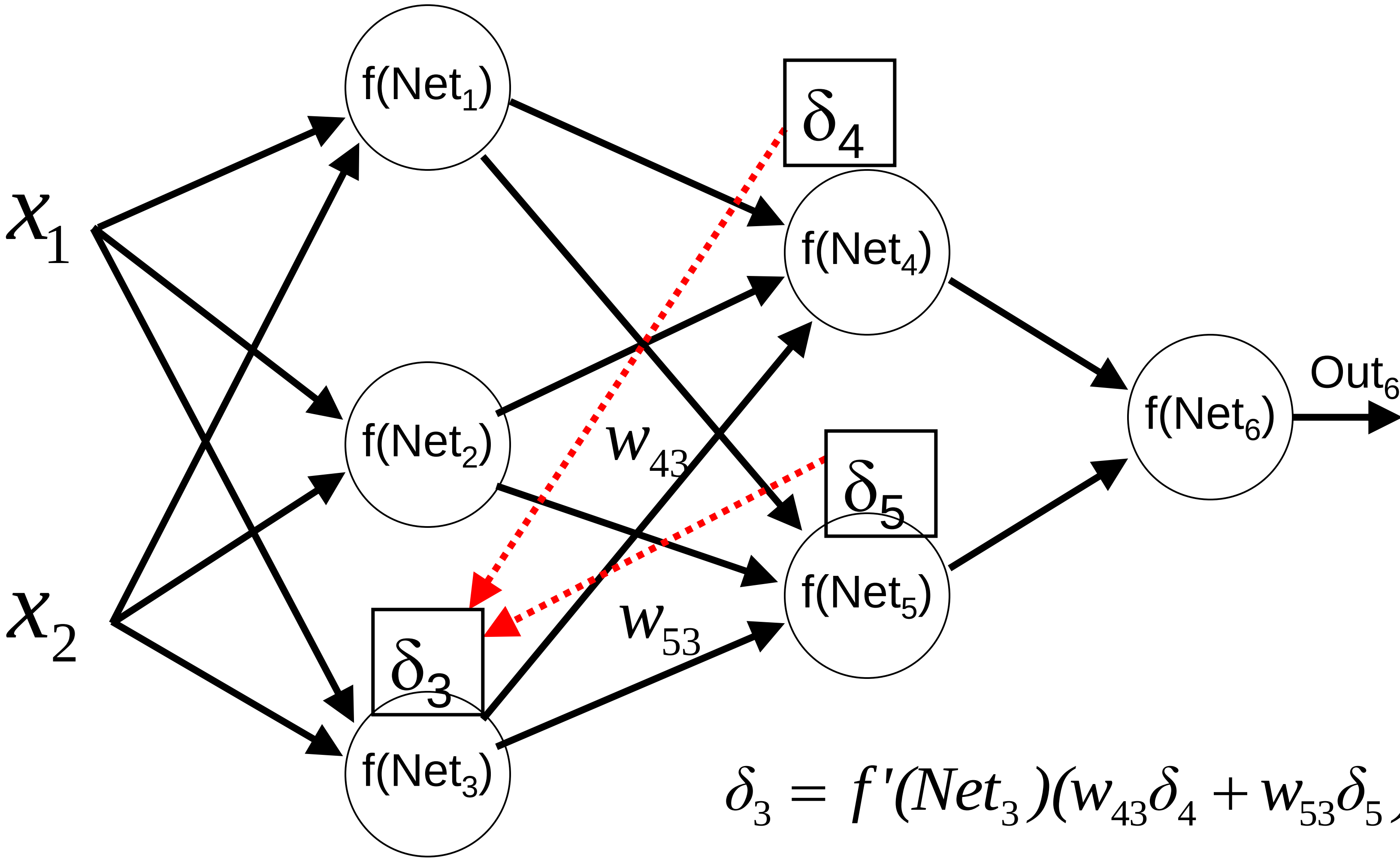
$$\delta_1 = f'(Net_1)(w_{41}\delta_4 + w_{51}\delta_5)$$

BP algorithm – Backward propagation (4)



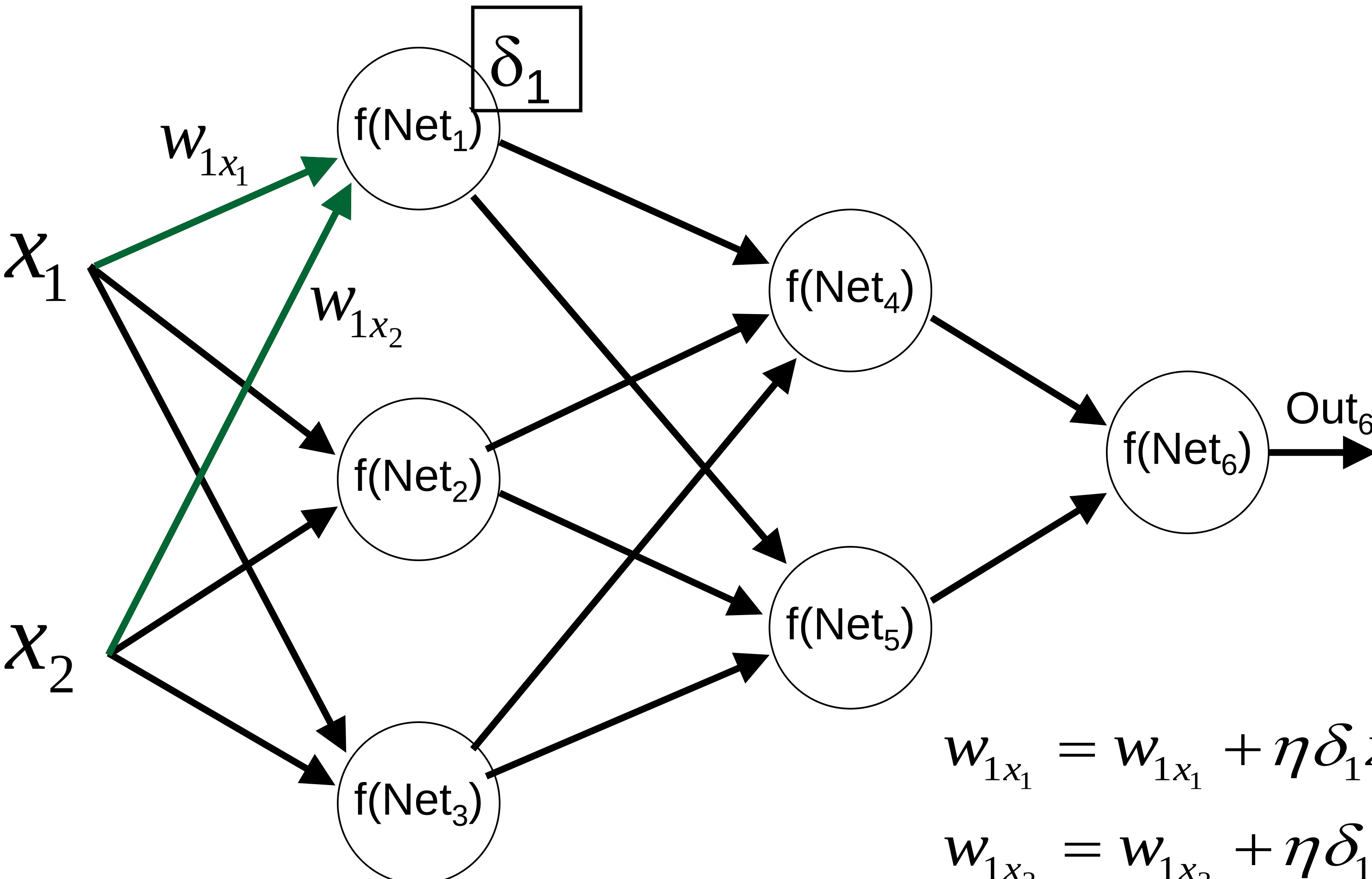
$$\delta_2 = f'(Net_2)(w_{42}\delta_4 + w_{52}\delta_5)$$

BP algorithm – Backward propagation (5)



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

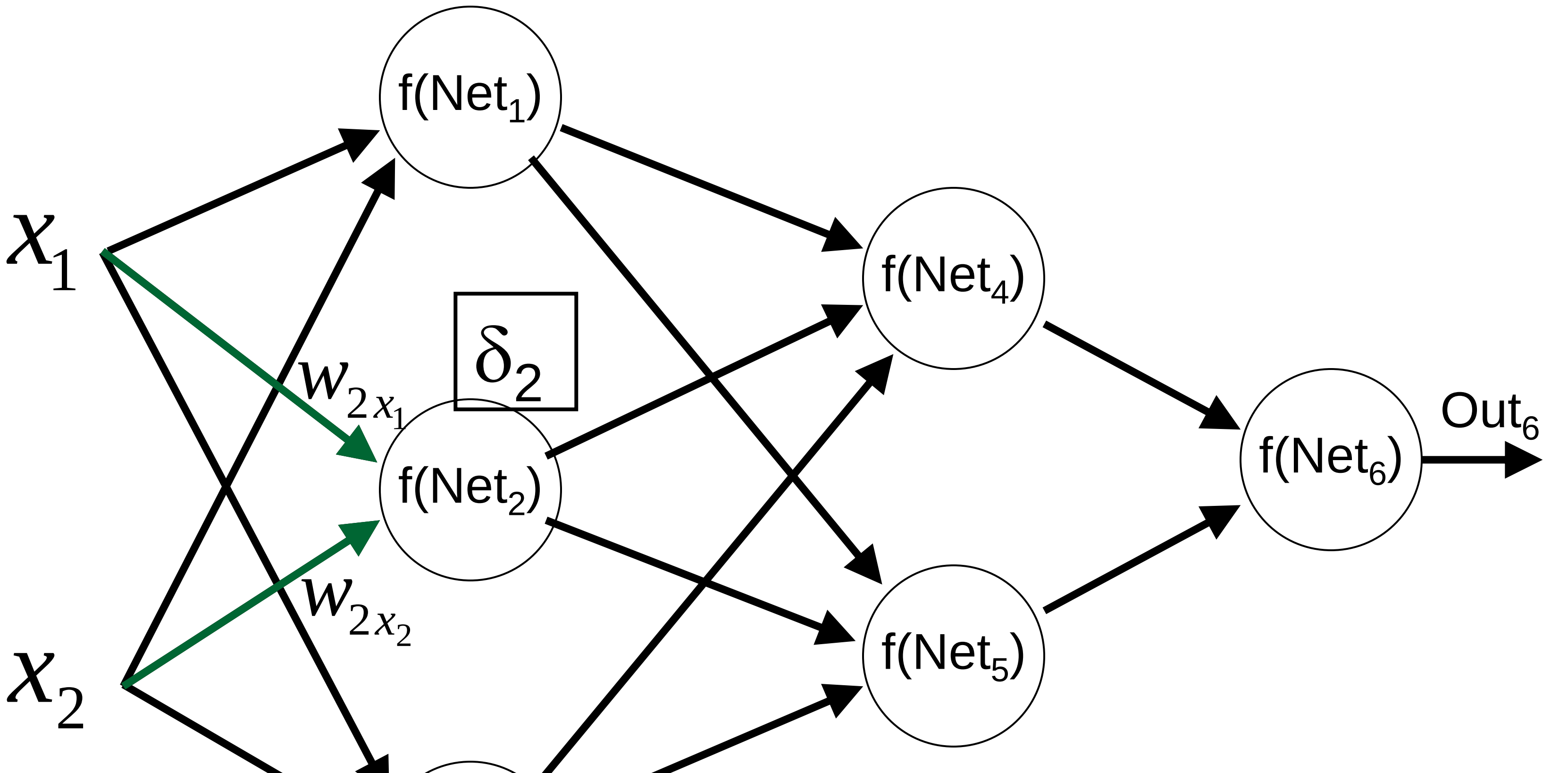
BP algorithm – Weight update (1)



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

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

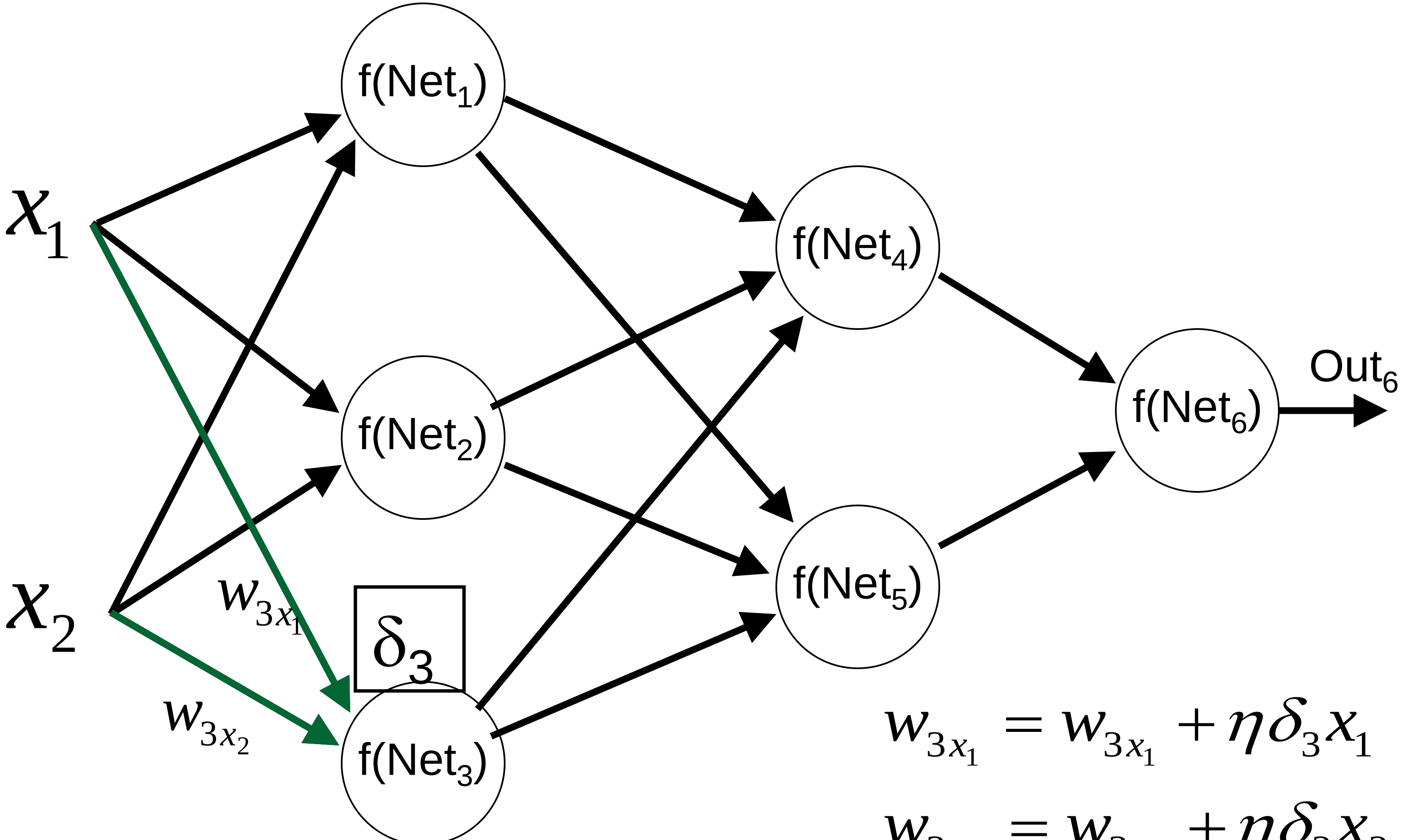
BP algorithm – Weight update (2)



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

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

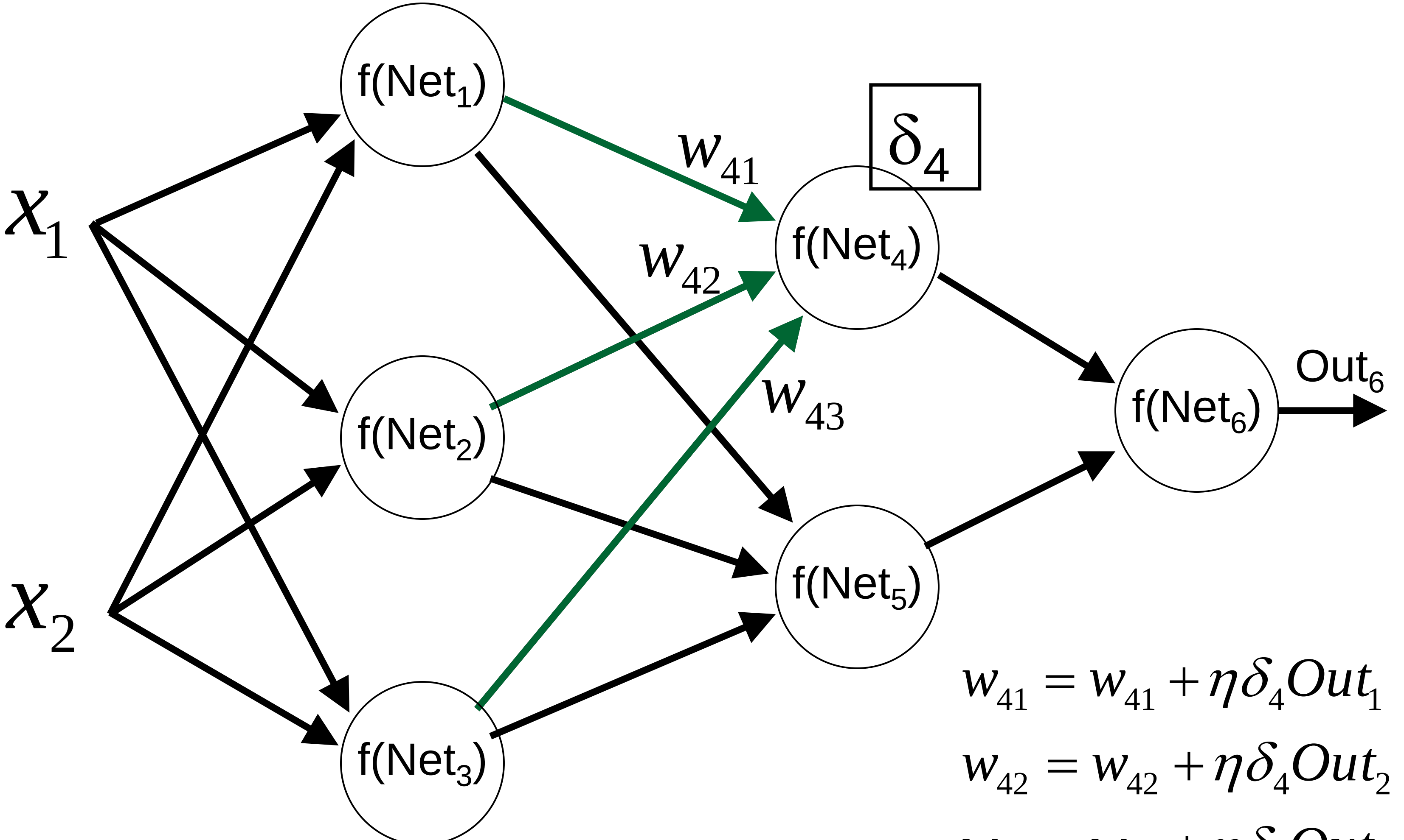
BP algorithm – Weight update (3)



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

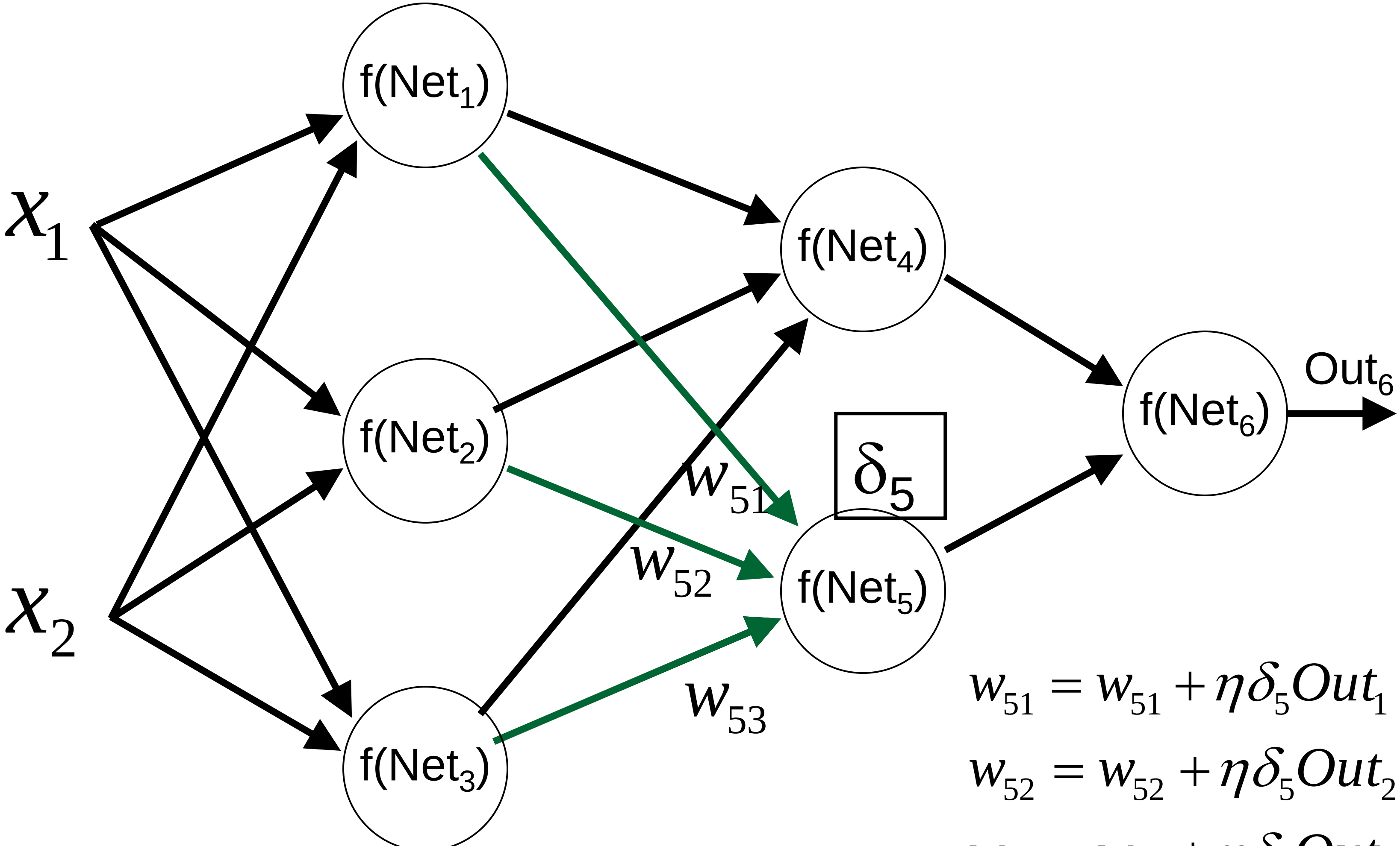
$$w_{3x_2} = w_{3x_2} + \eta\delta_3 x_2$$

BP algorithm – 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 algorithm – Weight update (5)

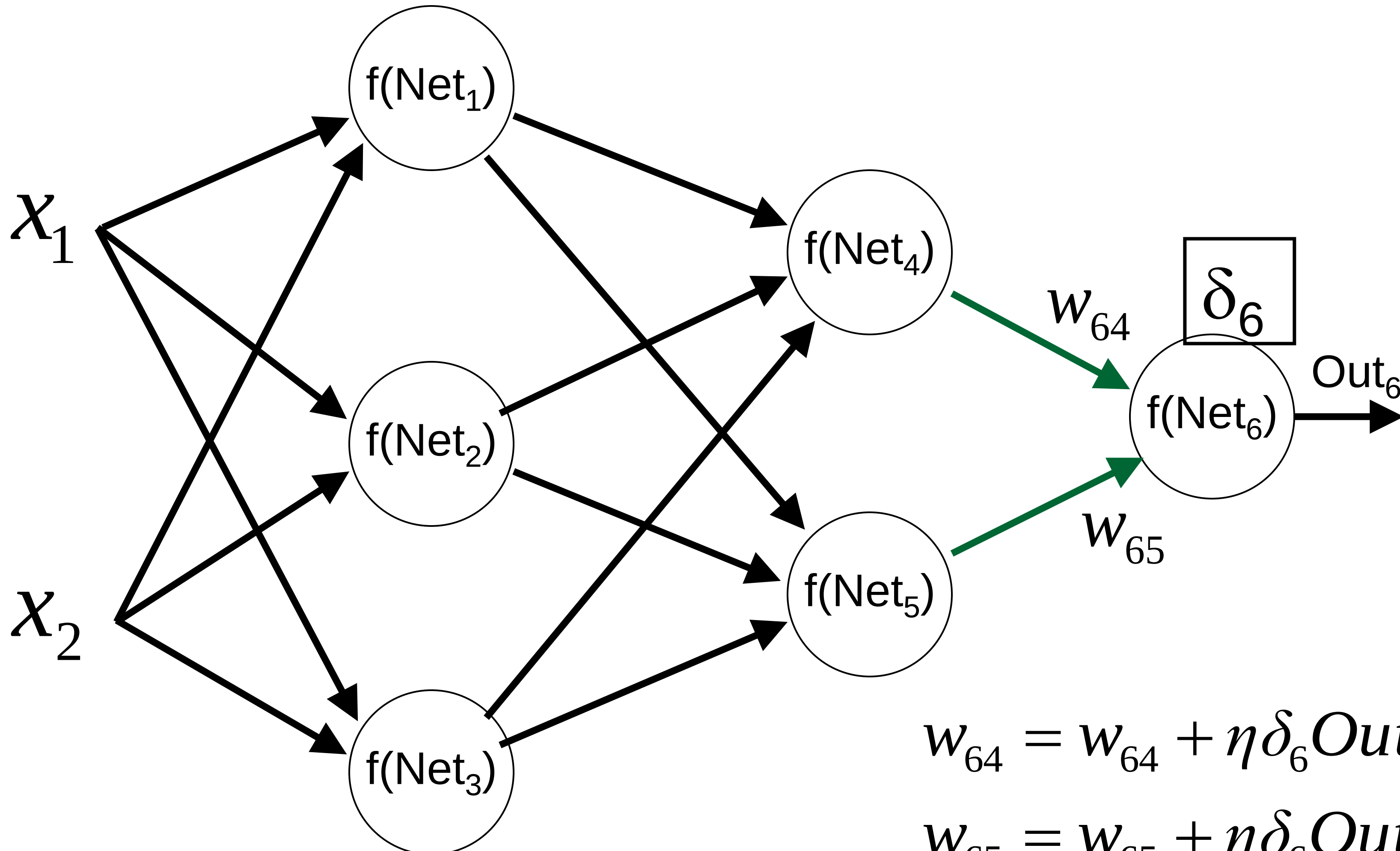


$$w_{51} = w_{51} + \eta\delta_5 Out_1$$

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

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

BP algorithm – Weight update (6)



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

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

BP: Initialization of weight values

- Often, the weights are initialized by small and random values
- If the weights have large initial values, then:
  - The sigmoid functions soon reaches saturation state
  - The system gets stuck at a local minimum or in a very flat plateau close to the starting point (of the training process)
- Suggestions for w^0_{ab} (of the link from neuron *b* to neuron *a*)
  - Let's call n_a is the number of neurons at the same layer of neuron *a*

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

  - Let's call k_a is the number of neurons that link to neuron *a* (=the number of the input links of neuron *a*)

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

BP: Learning rate

- Strongly influence the efficiency and convergence of the BP learning algorithm
  - A large value η may accelerate the convergence of the training process, but can make the system bypass the global optimal point or converge to a local optimal point
  - A small value η can make the training process (very) long
- Often the learning rate is selected *experimentally* for a specific problem
- A good learning rate at the beginning of the training process may become not good at a later time
  - We should employ an adaptive (i.e., dynamic) learning rate
- After updating the weights, check if the weights update reduces the total error

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

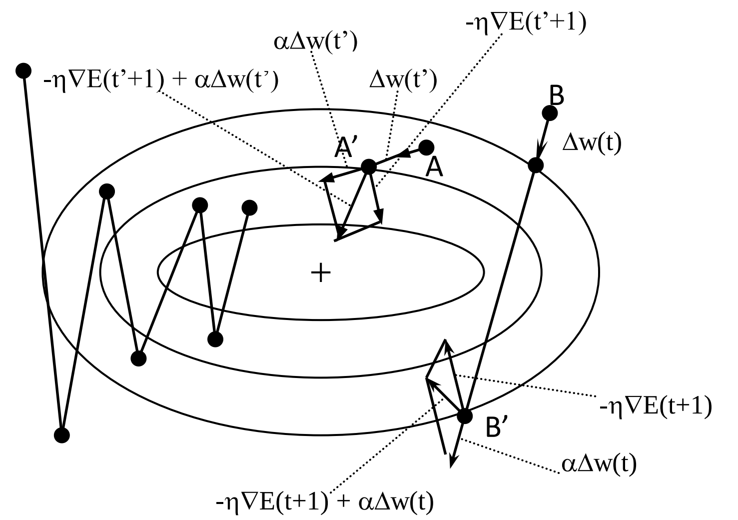
BP: Momentum

- The *Gradient descent* method can be too slow if η is small, and can oscillate much if η is too large
- To reduce the oscillation, we need to put in a new component called *momentum*

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

where α ($\in$[0,1]) is a momentum parameter (often selected =0.9)

- A hint, derived from experiments, to select appropriate values for the learning rate and momentum: $(\eta+\alpha) >\approx 1$; where $\alpha>\eta$ to avoid oscillation



Gradient descent for a simple second-order error function.

The left trajectory does not use momentum.

The right trajectory uses momentum.

BP: The number of neurons at a hidden layer

- The size (i.e., the number of neurons) of a hidden layer is an important question for the application of multi-layer feedforward ANNs to solve practical problems
- In practice, it is very difficult to determine precisely the number of neurons necessary to get an expected accuracy of the system
- The size of a hidden layer is often selected experimentally (i.e., trial and test)
- Recommendations:
  - Start with a small number of neurons at the hidden layer (=a small ratio compared to the number of input signals)
  - If the ANN cannot converge, then add more neurons to the hidden layer
  - If the ANN converges, then consider to reduce the number of neurons of the hidden layer

ANN – Advantages, Disadvantages

- Advantages:
  - It supports (in nature of the structure) parallel computation at a very high degree
  - Noise/error tolerance, by the parallel computing architecture
  - Can be designed to be self-adaptive (of weights, network structure)
- Disadvantages:
  - No general rule for determining the network structure and the training hyper-parameters' optimal values for a (family of) problem(s)
  - No general method for evaluating the internal operation of the ANN (therefore, the ANN system is considered a "black box")
  - Very difficult (if not impossible) to produce explanations for users
  - Very challenging to predict the system performance in future (i.e., the generalization capability of a learning system)

ANN – When?

- A training example is represented by set of (very) many discrete- or numeric-valued attributes
- The output value domain of the target function has a real or discrete or vector type
- The dataset may contain noise/error
- The representation (i.e., form) of the target function is unknown
- Not required (or not important) to show explanations for users for the (classification/prediction) results
- It is acceptable for a (very) long time of training
- Require for a (very) quick time of classification/prediction