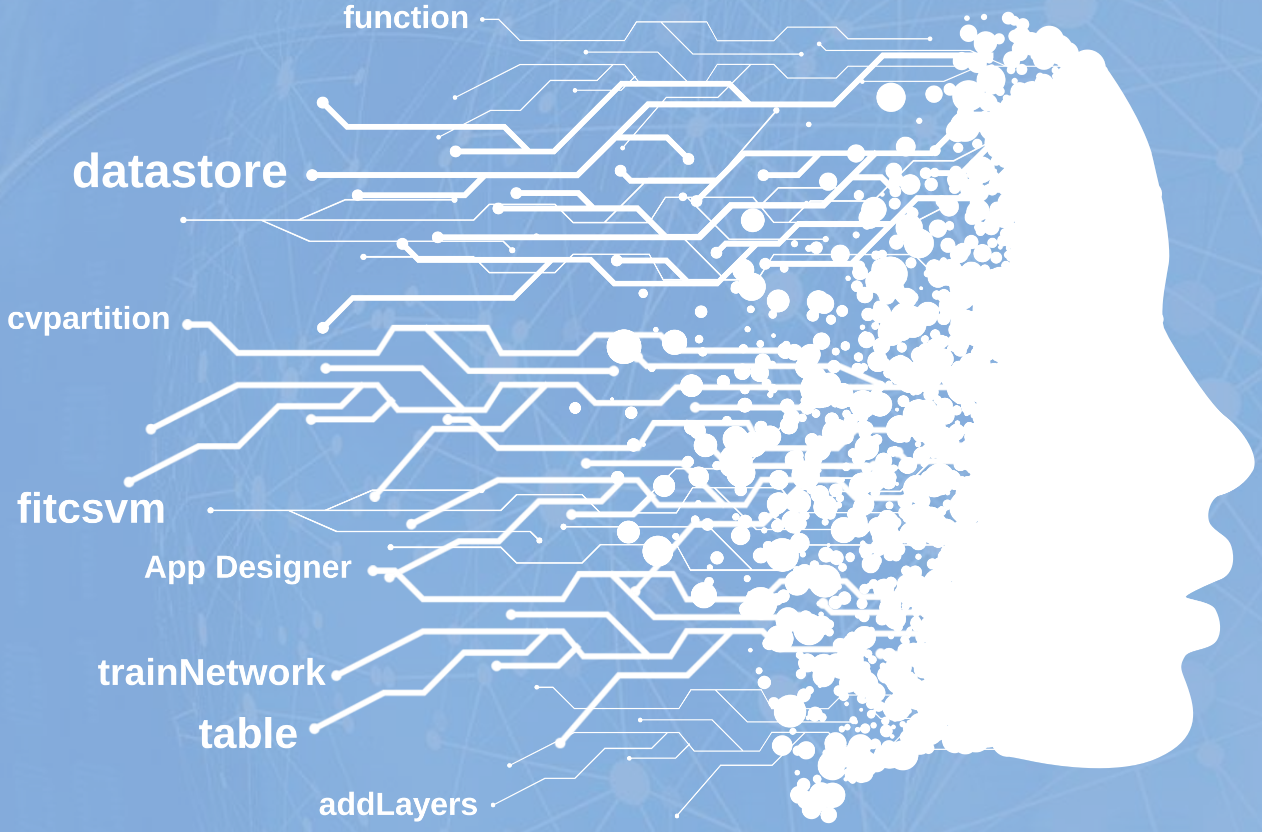
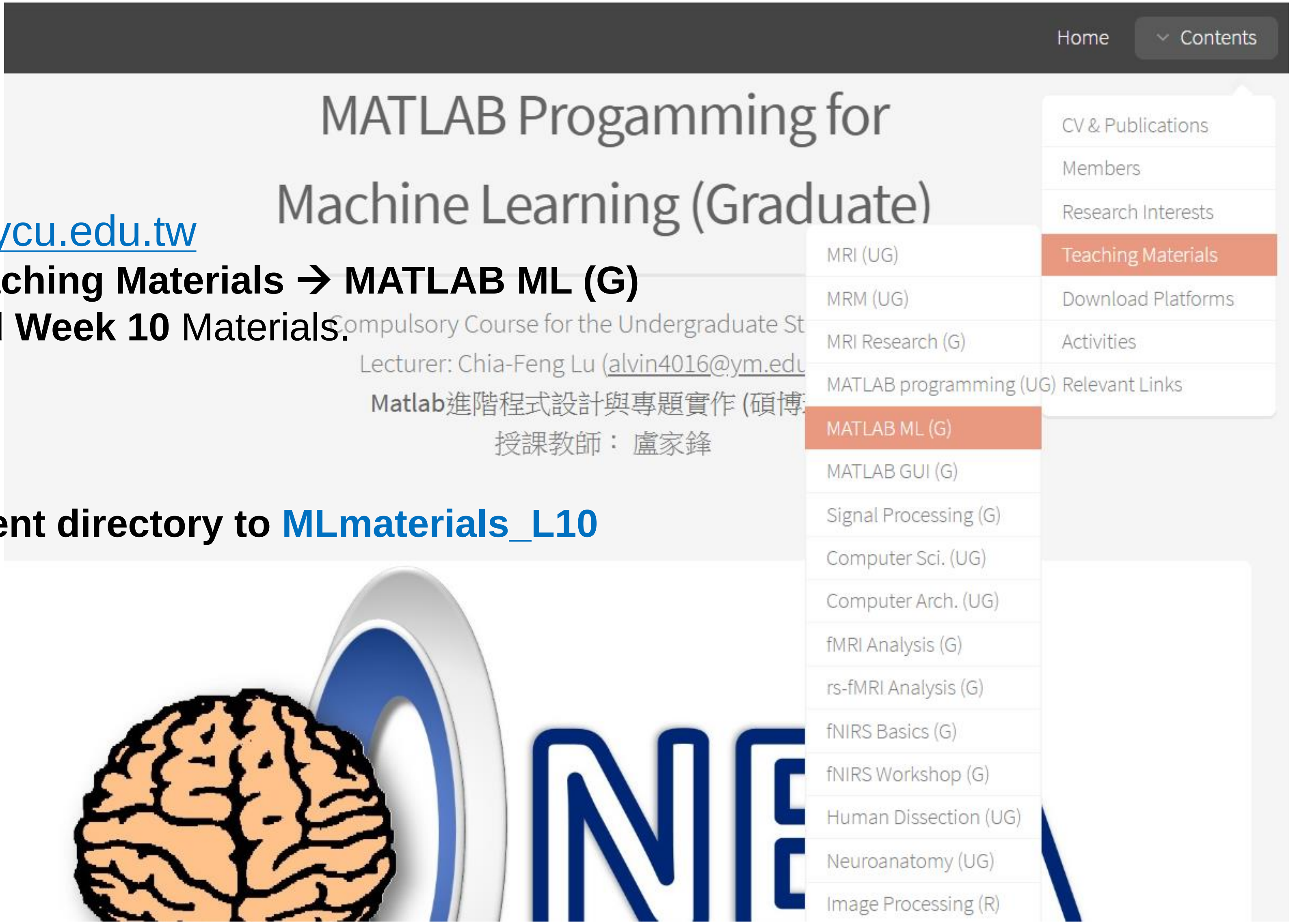


Neural Network

MATLAB進階程式語言與實作

盧家鋒 Chia-Feng Lu, Ph.D.
Department of Biomedical Imaging and Radiological Sciences, NYCU
alvin4016@nycu.edu.tw

Teaching Materials



http://cflu.lab.nycu.edu.tw

Contents → Teaching Materials → MATLAB ML (G)

Please download **Week 10** Materials.

Please set current directory to MLmaterials_L10

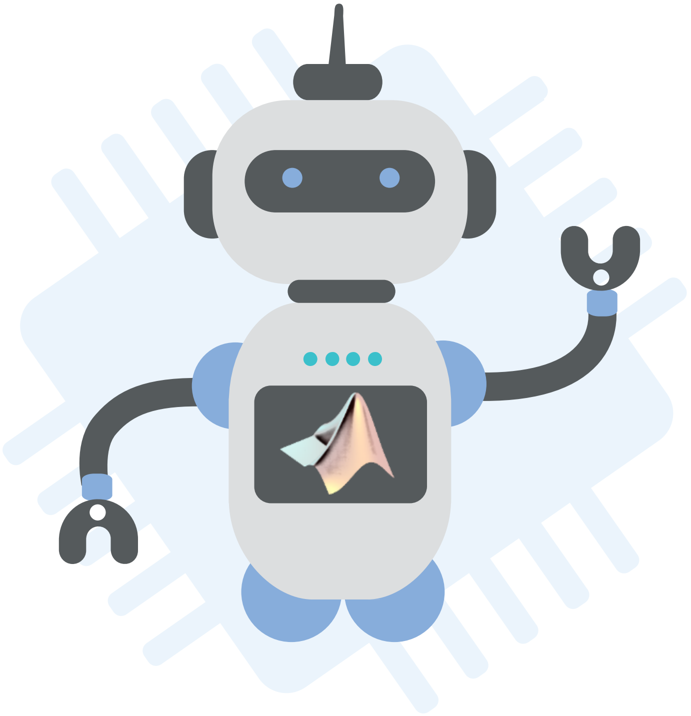
Contents in this Week

01 **Single-Layer Neural Network**

Basic Concepts and Supervised Learning

02 **Multi-Layer Neural Network**

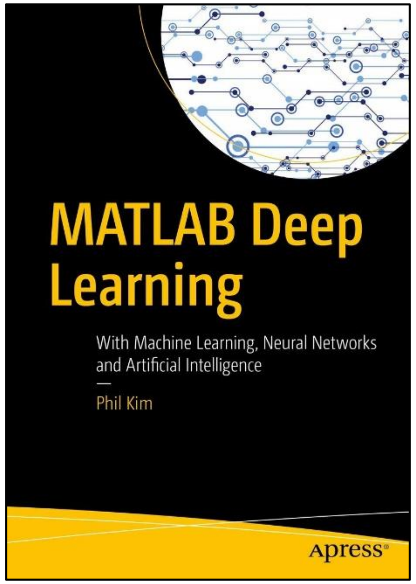
Back-propagation Algorithm, Momentum,
Cross Entropy, Regularization



References

[Textbook 5]

- **MATLAB Deep Learning With Machine Learning, Neural Networks and Artificial Intelligence, 1st edition, 2017**
  Phil Kim
- **Neural Network (Ch.2)**
- **Training of Multi-Layer Neural Network (Ch.3)**



Single-Layer Neural Network

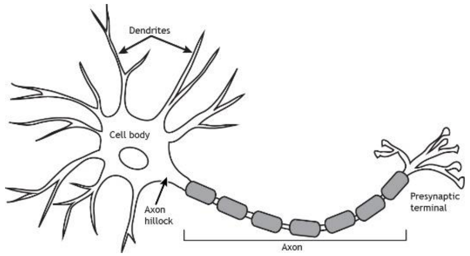
Basic Concepts and Supervised Learning

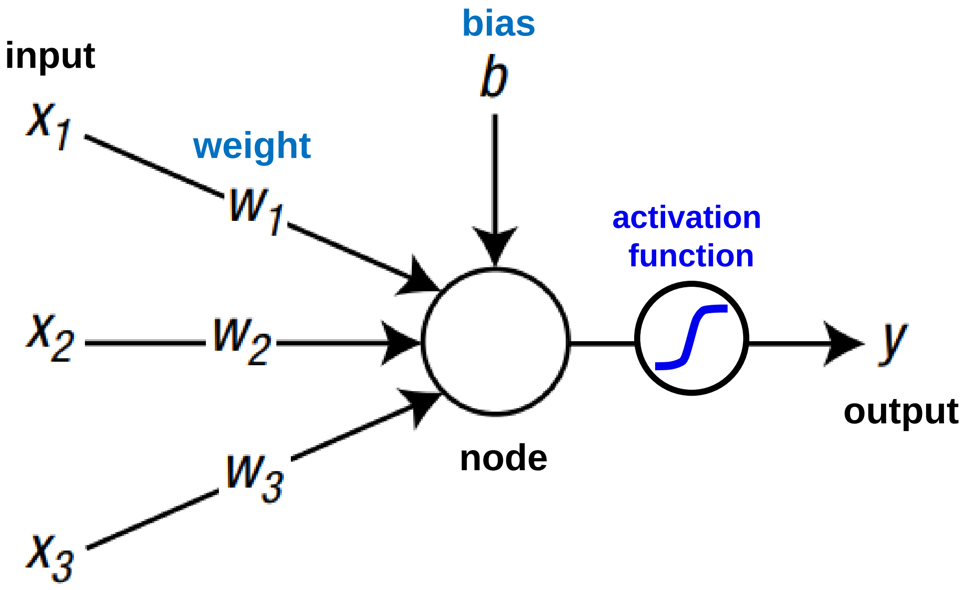
(Artificial) Neural Network

- The neural network imitates the mechanism of the brain. As the brain is composed of connections of numerous neurons

- The information of the neural net is stored in the form of weights and bias.

Brain	Neural Network
Neuron cell	Node
Connection of neurons	Connection weight
Action potential	Activation function

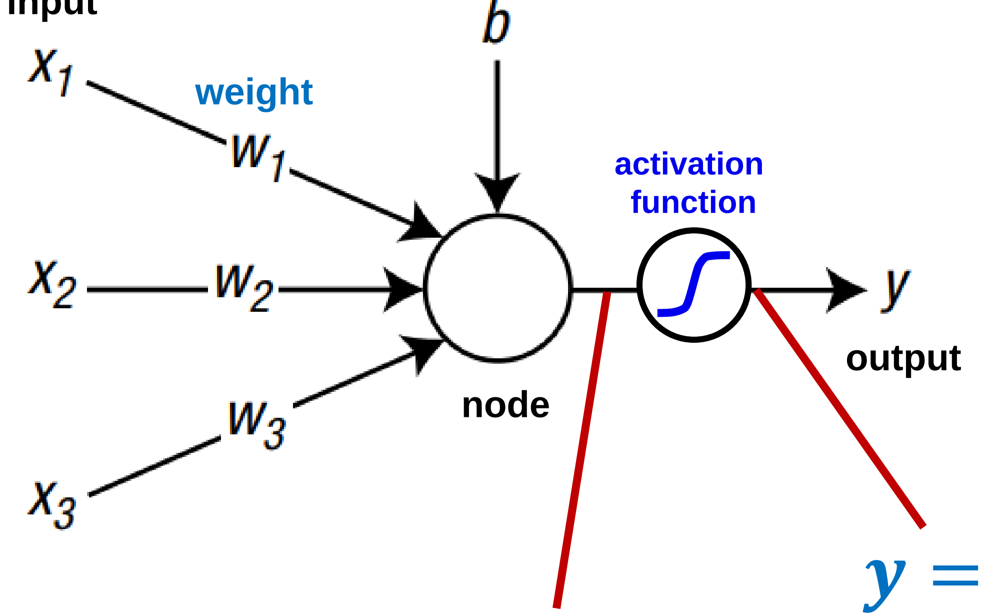




The arrows of the figure denote signal flow.

Neural Network

Mathematical representations:



$$y = \varphi(v)$$

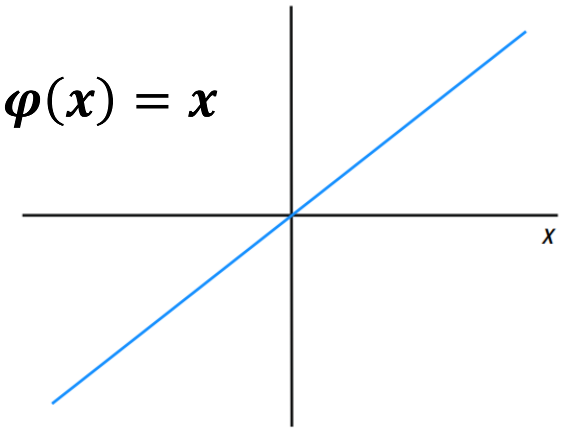
Linear or nonlinear transformation

$$v = WX + b$$

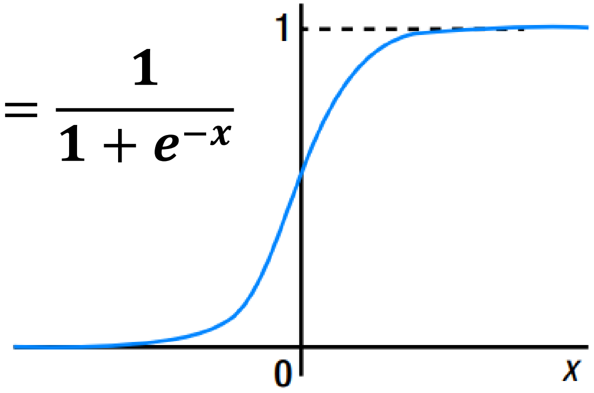
The weighted sum of inputs

linear function

$$\varphi(x) = x$$

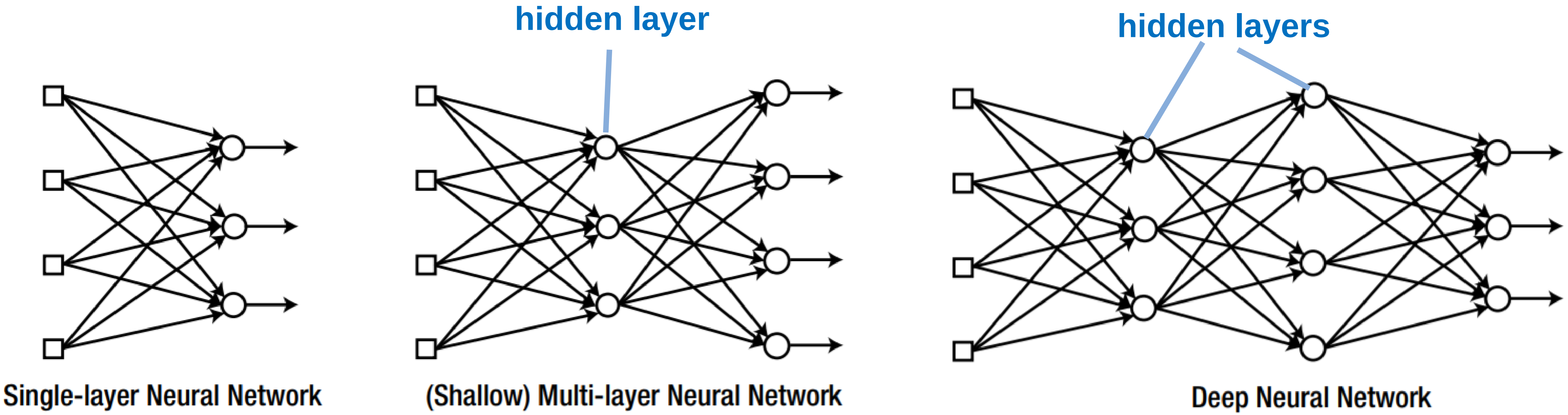


$$\varphi(x) = \frac{1}{1 + e^{-x}}$$



sigmoid function

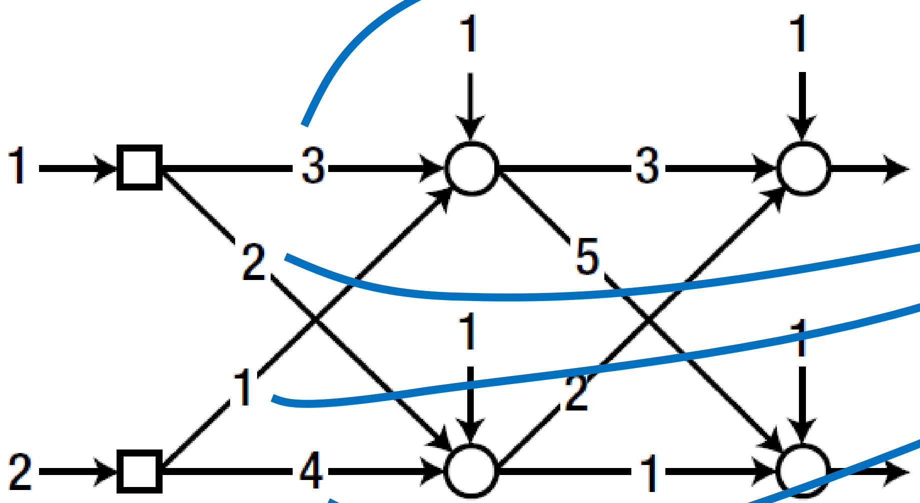
Layers of Neural Network



Single-Layer Neural Network		Input Layer – Output Layer
Multi-Layer Neural Network	Shallow Neural Network	Input Layer – Hidden Layer – Output Layer
	Deep Neural Network	Input Layer – Hidden Layers – Output Layers

Why nonlinear activation function?

- The use of a linear function for the nodes negates the effect of adding a layer.



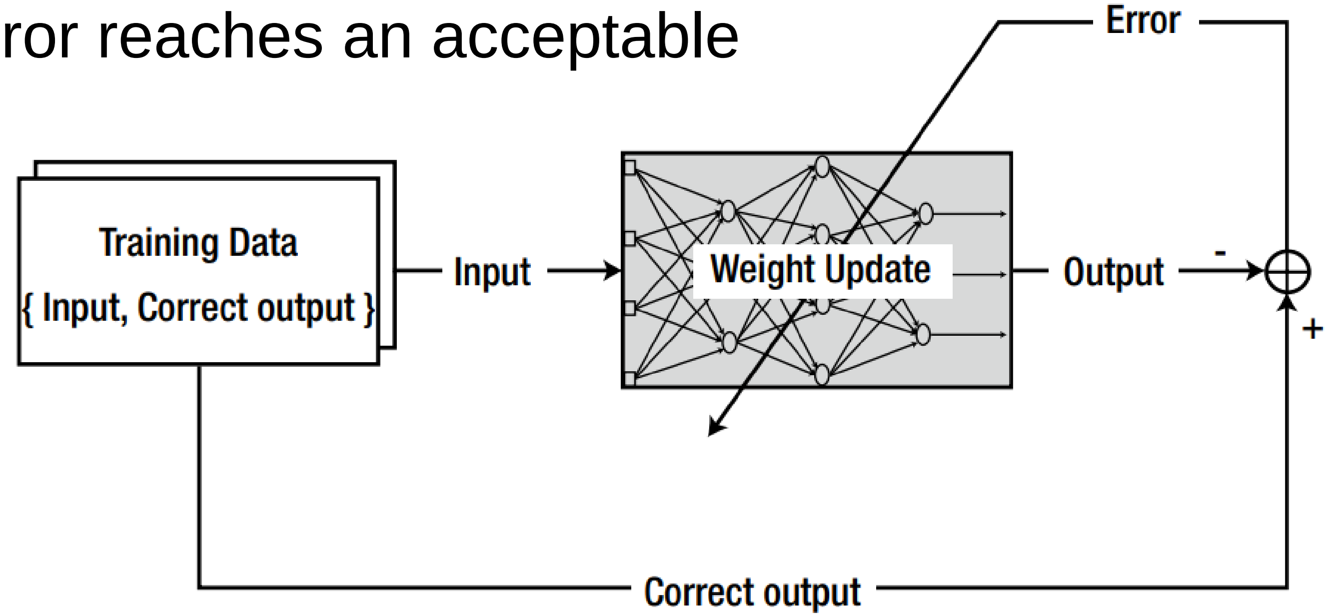
$$
\begin{aligned}
v &= \begin{bmatrix} 3 & 2 \\ 5 & 1 \end{bmatrix} \left(\begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 3 & 2 \\ 5 & 1 \end{bmatrix} \begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 3 & 2 \\ 5 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 13 & 11 \\ 17 & 9 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 6 \\ 7 \end{bmatrix}
\end{aligned}
$$

- In this case, the multi-layer model is mathematically identical to a single-layer neural network.

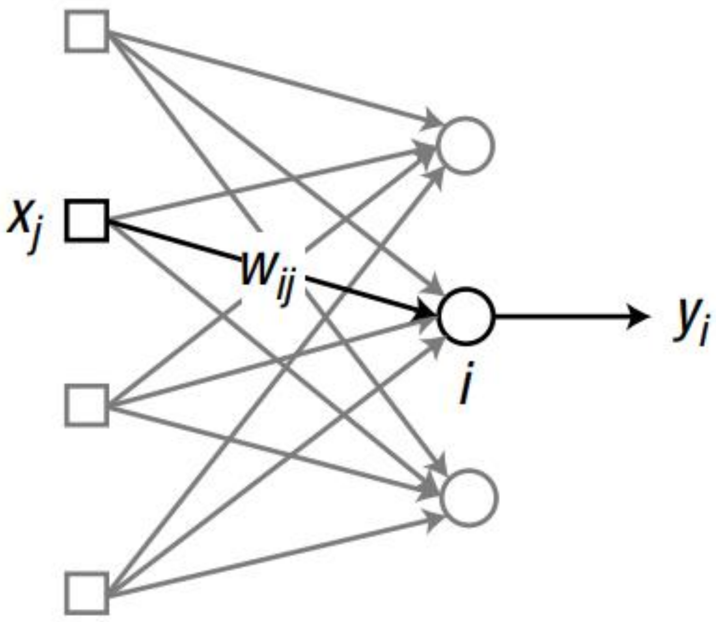
Supervised Learning of a Neural Network

1. **Initialize** the weights with adequate values.
2. Enter the training data { input, correct output } into the neural network, and **calculate the error from the correct output**.
3. **<u>Adjust</u> the weights to reduce the error**.
4. Repeat Steps 2-3 for all training data
5. Repeat Steps 2-4 until the error reaches an acceptable tolerance level.

(All training data goes through Steps 2-4 once, is called an **epoch**.)



Errors and Loss Function



$$e_i = d_i - y_i$$

d_i is the correct output of the output node *i*.
(ground truth)

- Let us define the loss function for output node y_i

$$L_i = \frac{1}{2}(d_i - y_i)^2, \qquad y_i = \varphi(v_i),\ v_i = \sum_{j=1}^{m} w_{ij}x_j$$

where *m* is the numbers of input nodes

Steepest Gradient Decent

- We minimize the loss function L_i w.r.t w_{ij}

$$\frac{\partial L_i}{\partial w_{ij}} = e_i(-1)\frac{\partial \varphi}{\partial v_i}\frac{\partial v_i}{\partial w_{ij}} = -e_i\varphi' x_j$$

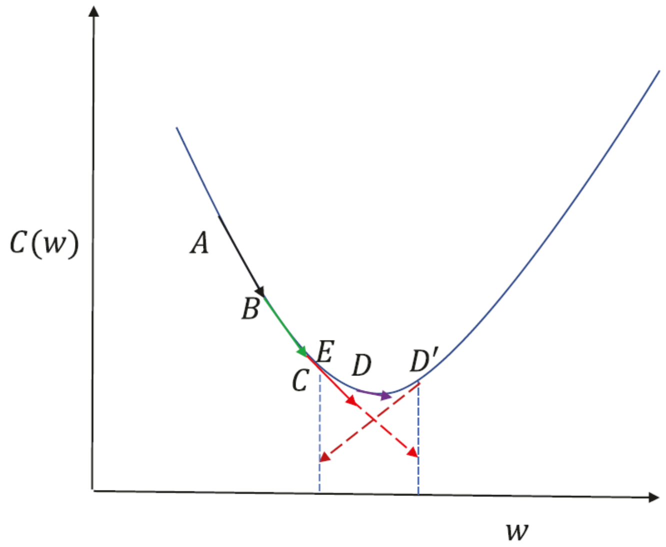
- The steepest gradient decent method

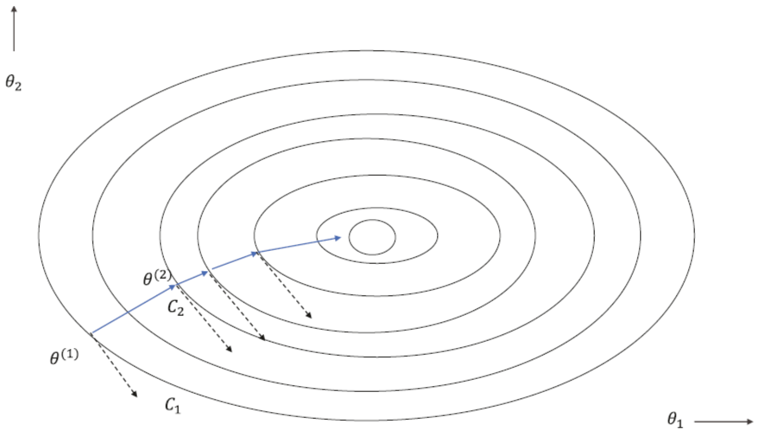
$$w_{ij}^{(k+1)} = w_{ij}^{(k)} - \alpha\frac{\partial L}{\partial w_{ij}}$$

$$= w_{ij}^{(k)} + \alpha\varphi' e_i x_j$$

- Or we may express the above equation as

$$w_{ij} \leftarrow w_{ij} + \alpha\varphi' e_i x_j$$





Generalized Delta Rule

- For an arbitrary activation function, the delta rule is expressed as

$$\boldsymbol{w_{ij} \leftarrow w_{ij} + \alpha\delta_i x_j}$$

$$\boldsymbol{\delta_i = \varphi'(v_i)e_i}$$

- The weight is adjusted in proportion to the input value, x_j and the output error, e_i.

w_{ij} = The weight between the output node i and input node j
e_i = The error of the output node *i*
v_i = The weighted sum of the output node *i*
φ' = The derivative of the activation function φ of the output node *I*
α = Learning rate ($0 < \alpha \leq 1$)

Derivative of sigmoid function

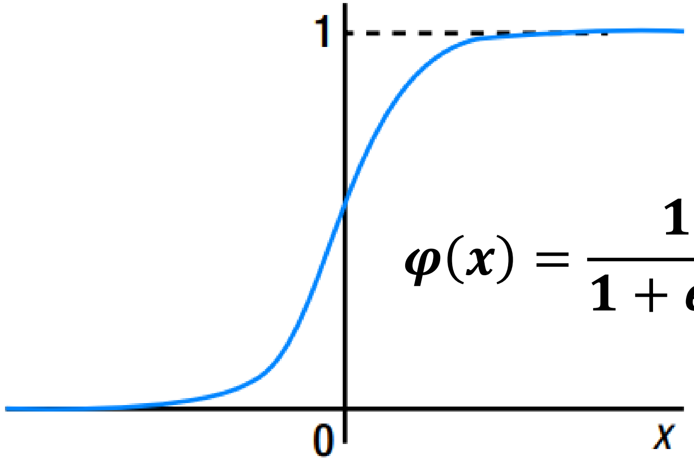
$$\varphi'(x) = \frac{d(1+e^{-x})^{-1}}{dx} = -(1+e^{-x})^{-2}(-e^{-x})$$

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

$$= \varphi(x)(1-\varphi(x))$$

$$\boxed{\boldsymbol{w_{ij} \leftarrow w_{ij} + \alpha\varphi(x)(1-\varphi(x))e_i x_j}}$$

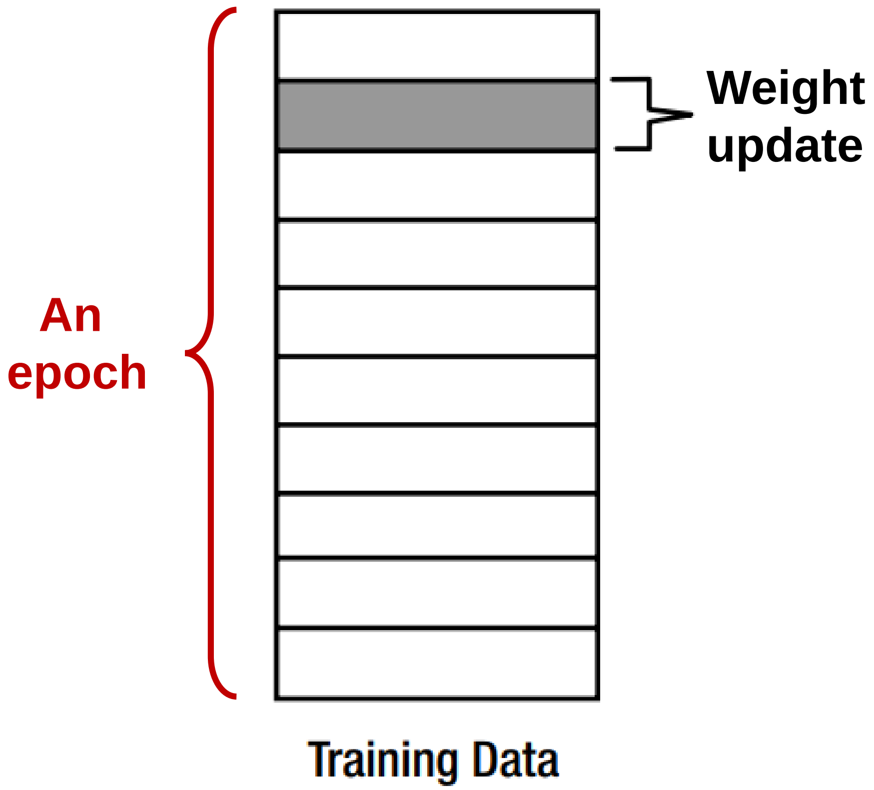
sigmoid function



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

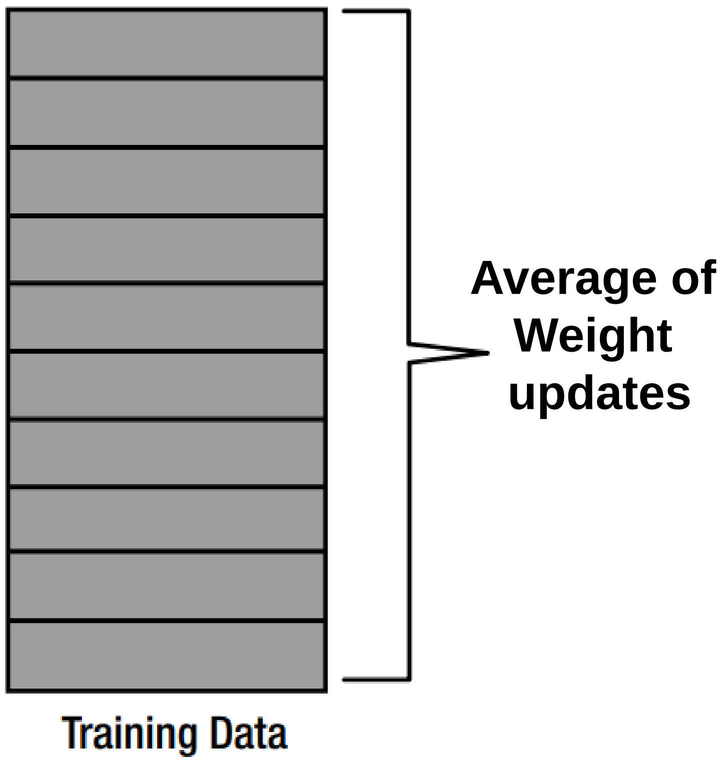
Schemes for Updating weights

Stochastic Gradient Descent (SGD)



Calculating the error for each training data and adjusts the weights immediately.

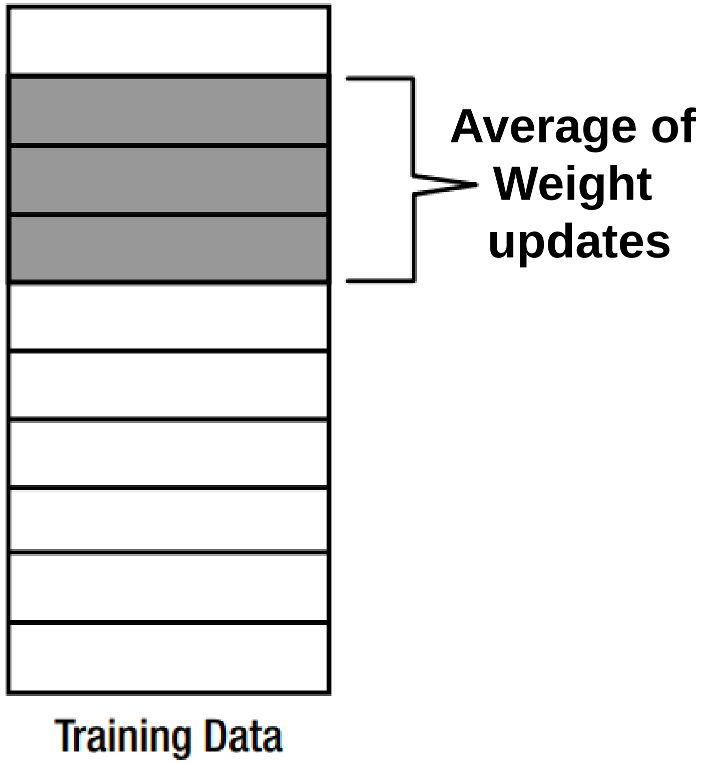
Batch



Each weight update is calculated for all errors of the training data.

http

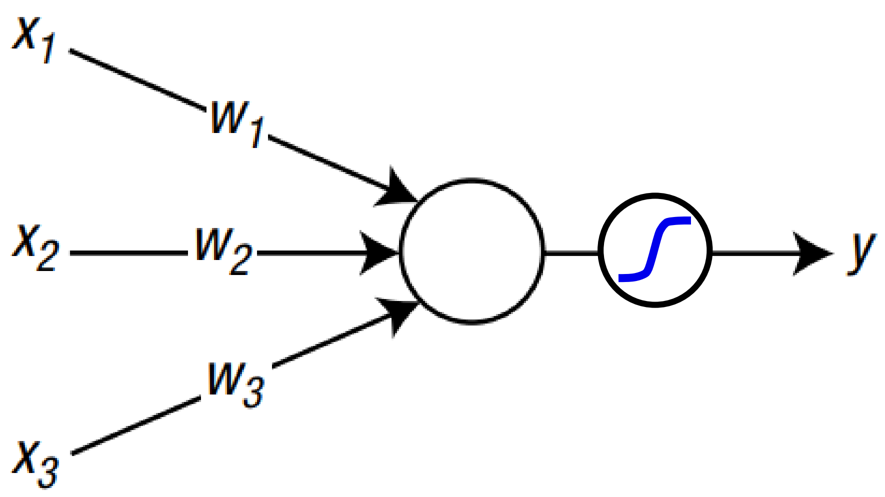
Mini Batch



It has speed from the SGD and stability from the batch.
It is often utilized in Deep Learning.

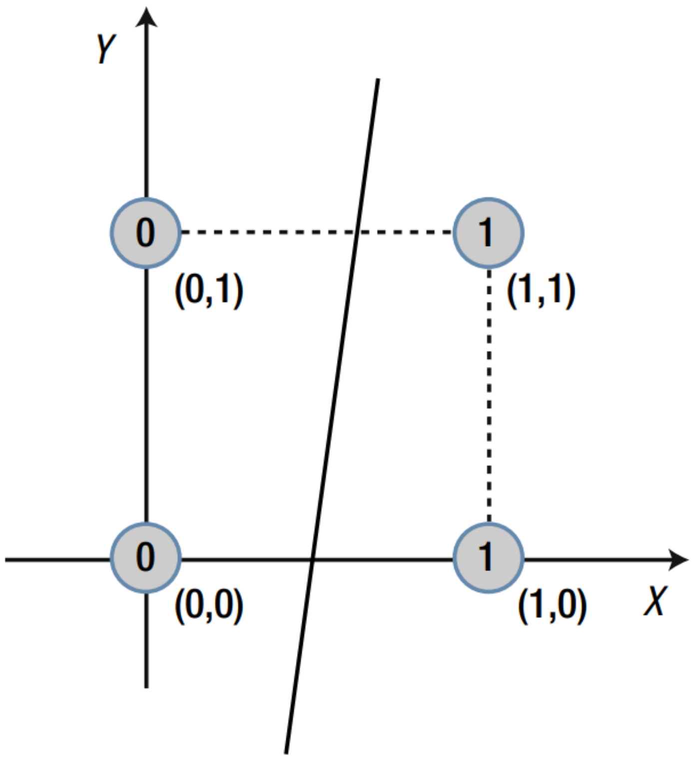
Example 1: Linearly Separable

- **MLmaterials_L10\Single-layer**
  - **TestDeltaSGD.m**
  - **DeltaSGD.m**
  - **Sigmoid.m**



responses

{0, 0, 1, 0}
{0, 1, 1, 0}
{1, 0, 1, 1}
{1, 1, 1, 1}



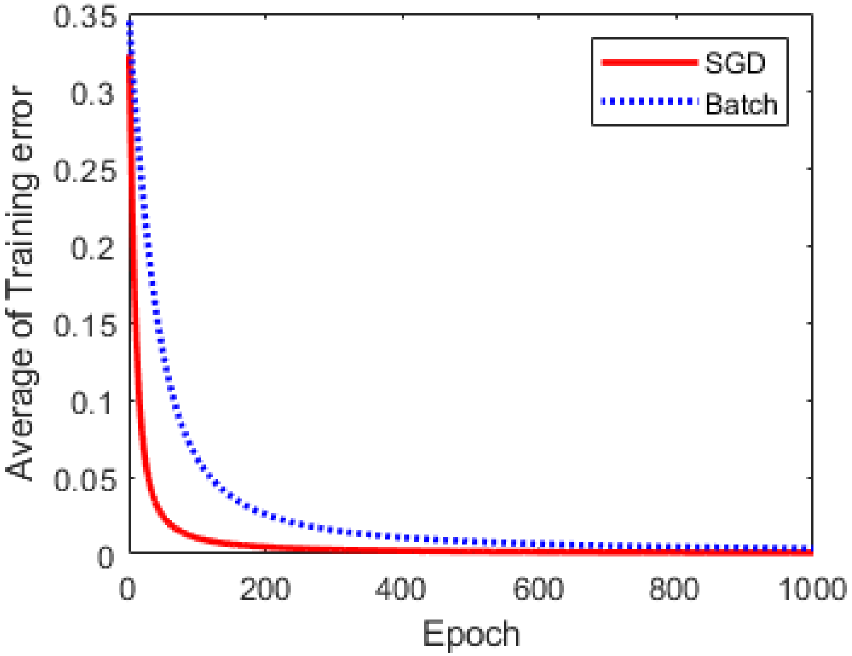
SGD vs. Batch

- **TestDeltaSGD.m**
- **DeltaSGD.m**

```
    v = W*x;
    y = Sigmoid(v);

    e     = d - y;
    delta = y*(1-y)*e;

    dW = alpha*delta*x;

    W(1) = W(1) + dW(1);
    W(2) = W(2) + dW(2);
    W(3) = W(3) + dW(3);
  end
end
```

- **TestDeltaBatch.m**
- **DeltaBatch.m**

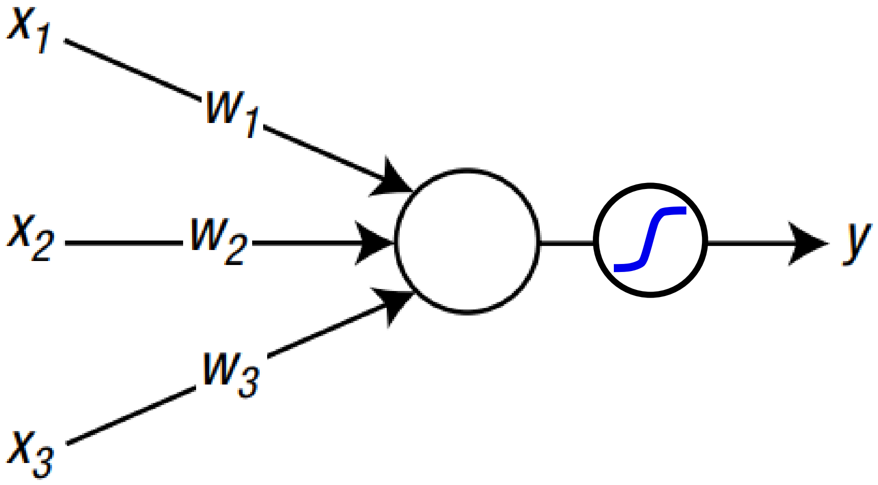
```
    v = W*x;
    y = Sigmoid(v);

    e     = d - y;
    delta = y*(1-y)*e;

    dW = alpha*delta*x;

    dWsum = dWsum + dW;
  end
  dWavg = dWsum / N;

  W(1) = W(1) + dWavg(1);
  W(2) = W(2) + dWavg(2);
  W(3) = W(3) + dWavg(3);
end
```



Average of Weight updates

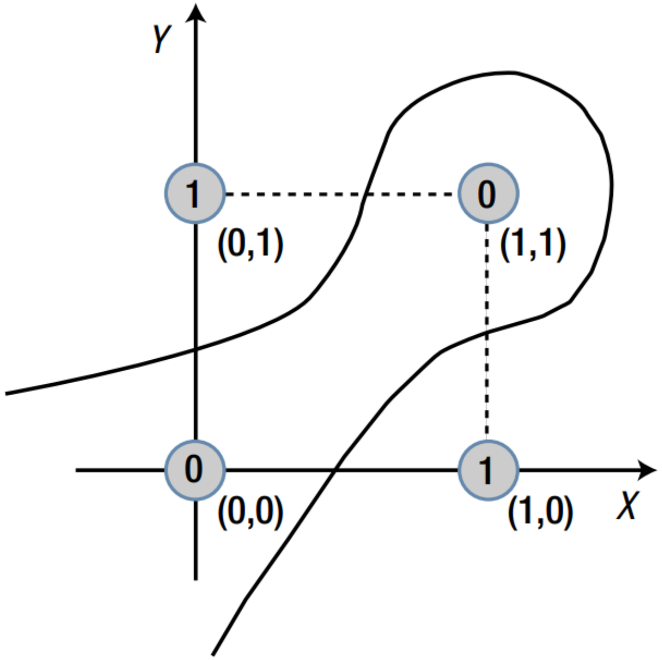
Example 2: Linearly Inseparable

- **MLmaterials_L10\Single-layer**
  - **TestDeltaSGD.m**
  - **DeltaSGD.m**
  - **Sigmoid.m**



responses

{0, 0, 1, **0**}
{0, 1, 1, **1**}
{1, 0, 1, **1**}
{1, 1, 1, **0**}



Short Summary

- The single-layer neural network can only solve linearly separable problems. This is because the single-layer neural network is a model that linearly divides the input data space.

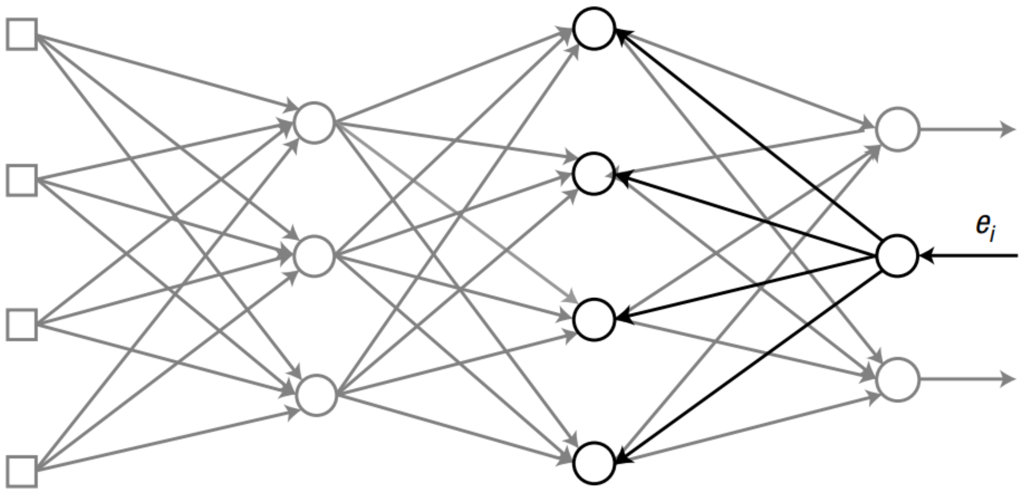
- In order to overcome this limitation of the single-layer neural network, **we need more layers** in the network.

Multi-Layer Neural Network

Back-propagation Algorithm, Momentum, Cross Entropy, Regularization

Back-Propagation Algorithm

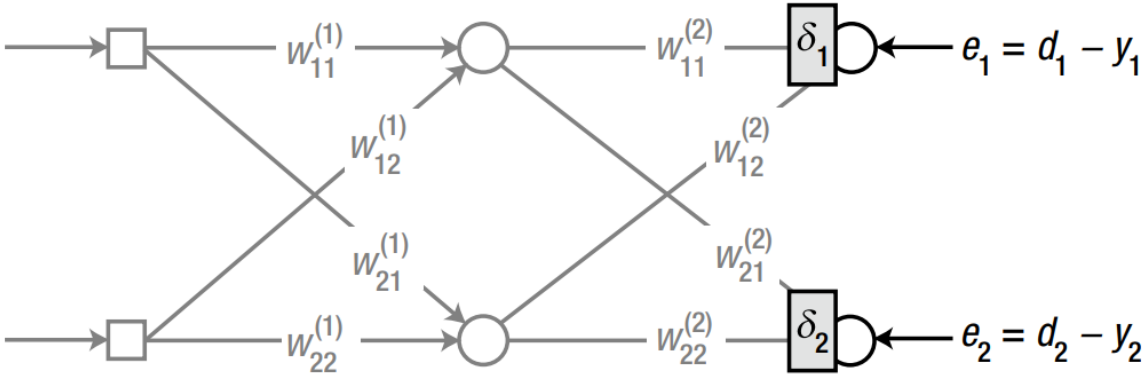
- The previously introduced delta rule is ineffective for training of the multi-layer neural network because the error is not defined in the hidden layers.
- **Back-propagation algorithm** provided a systematic method to determine the error of the hidden nodes.
- Once the hidden layer errors are determined, the delta rule is applied to adjust the weights.



the output error starts from the output layer and **moves backward** until it reaches the right next hidden layer to the input layer.

Back-Propagation Algorithm

- The first thing to calculate is delta, δ, of each node：

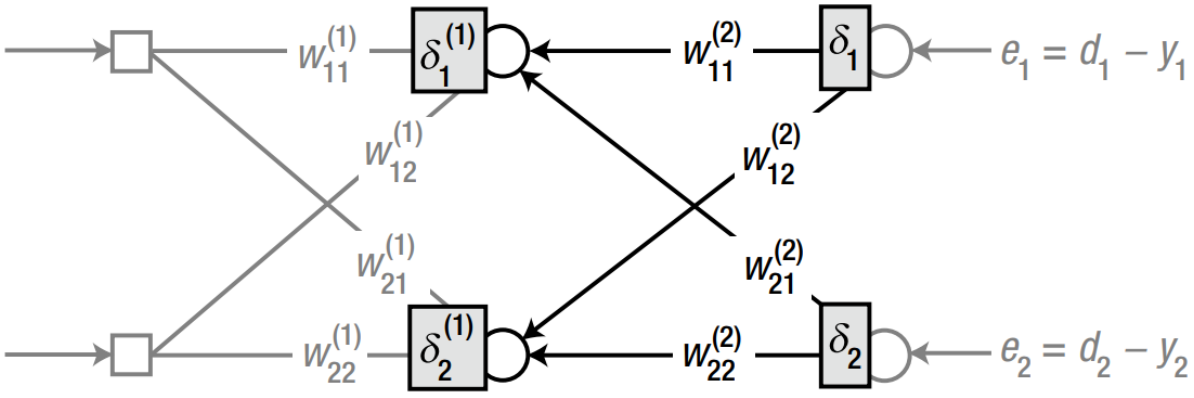


$$e_1 = d_1 - y_1 \qquad\qquad e_2 = d_2 - y_2$$
$$\delta_1 = \varphi'(v_1)e_1 \qquad\qquad \delta_2 = \varphi'(v_2)e_2$$

φ' is the derivative of the activation function of the output node.
y_i is the output from the output node.
d_i is the correct output from the training data.
v_i is the weighted sum of the corresponding node.

Back-Propagation Algorithm

- Since we have δ_1 and δ_2, let's proceed leftward to the hidden nodes and calculate the delta：



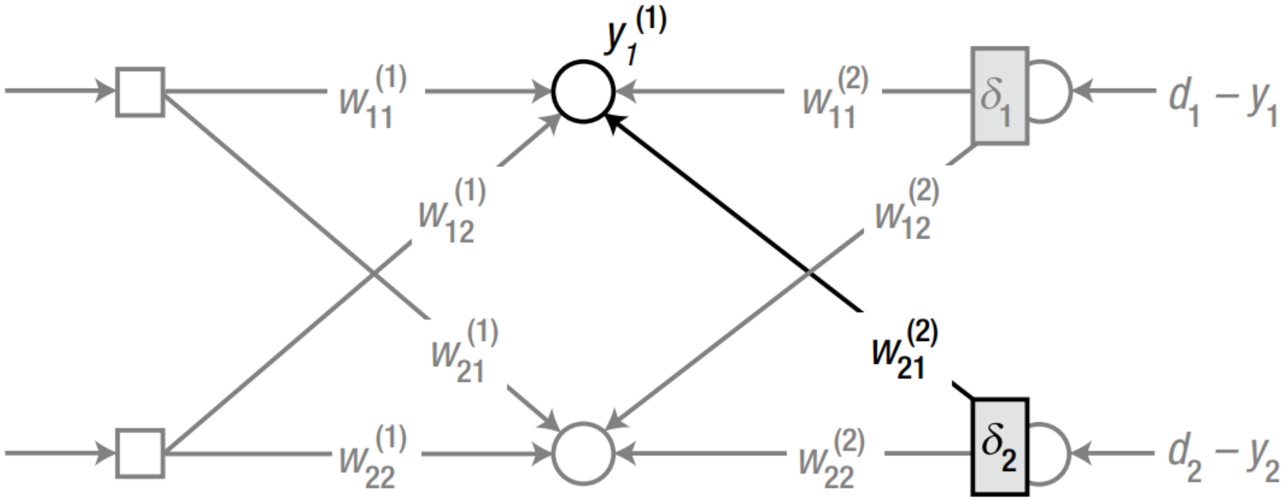
$$e_1^{(1)} = w_{11}^{(2)}\delta_1 + w_{21}^{(2)}\delta_2$$
$$\delta_1^{(1)} = \varphi'\left(v_1^{(1)}\right)e_1^{(1)}$$
$$= \varphi\left(v_1^{(1)}\right)(1-\varphi\left(v_1^{(1)}\right))e_1^{(1)}$$

$$e_2^{(1)} = w_{12}^{(2)}\delta_1 + w_{22}^{(2)}\delta_2$$
$$\delta_2^{(1)} = \varphi'\left(v_2^{(1)}\right)e_2^{(1)}$$
$$= \varphi\left(v_2^{(1)}\right)(1-\varphi\left(v_2^{(1)}\right))e_2^{(1)}$$

$v_1^{(1)}$ and $v_2^{(1)}$ are the weighted sums of the forward signals at the respective nodes.

Update of Weights

- Consider the weight $w_{21}^{(2)}$ for example.

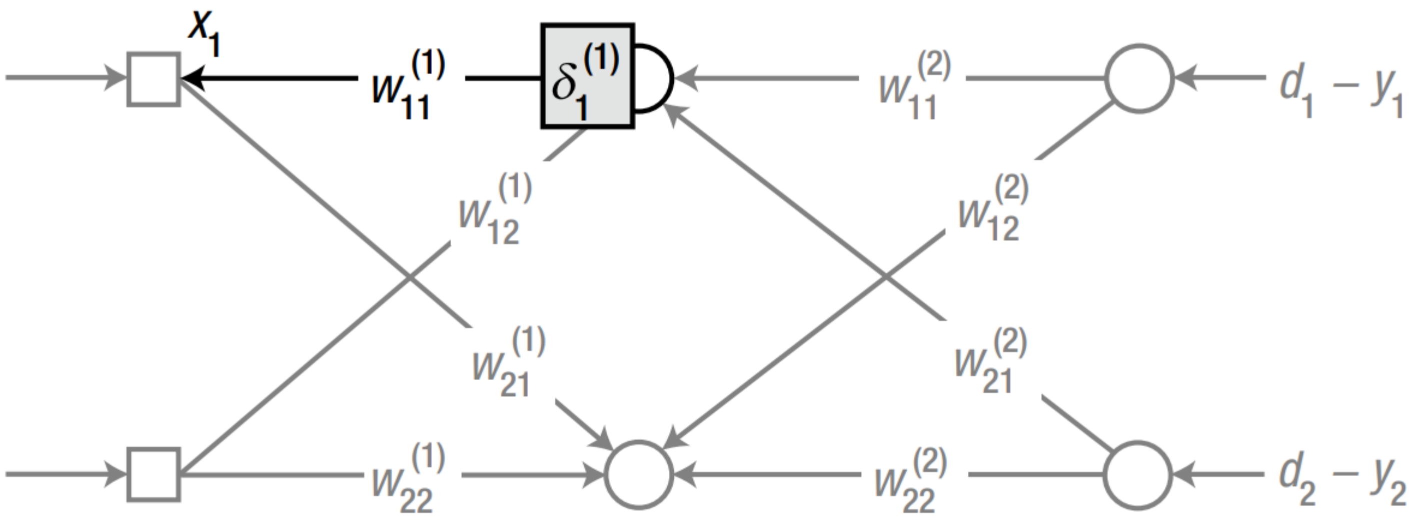


$$w_{21}^{(2)} \leftarrow w_{21}^{(2)} + \alpha \delta_2 y_1^{(1)}$$

$y_1^{(1)}$ is the output of the first hidden node.

Update of Weights

- The weight $w_{11}^{(1)}$ of figure is adjusted as：

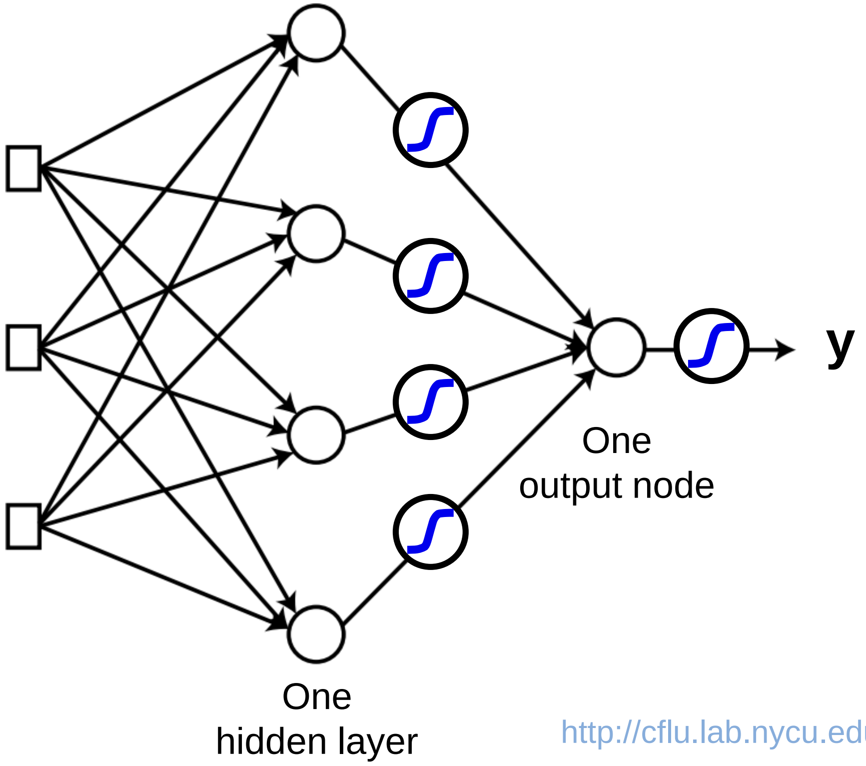


$$w_{11}^{(1)} \leftarrow w_{11}^{(1)} + \alpha \delta_1^{(1)} x_1$$

x_1 is the output of the first input node.

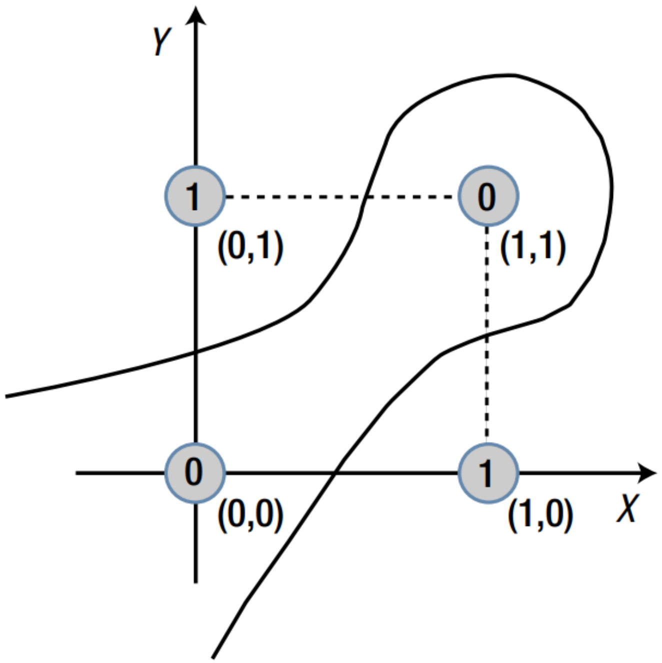
Example: Back-Propagation

- **MLmaterials_L10\Multi-layer**
  - **TestBackpropXOR.m**
  - **BackpropXOR.m**
  - **Sigmoid.m**



responses

{0, 0, 1, **0**}
{0, 1, 1, **1**}
{1, 0, 1, **1**}
{1, 1, 1, **0**}



Momentum

- The momentum, ***m***, is a term that is added to the delta rule for adjusting the weight.
- It acts similarly to physical momentum, which impedes the reaction of the body to the external forces.

$$\Delta w_{ij} = \alpha \delta_i x_j$$
$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$$

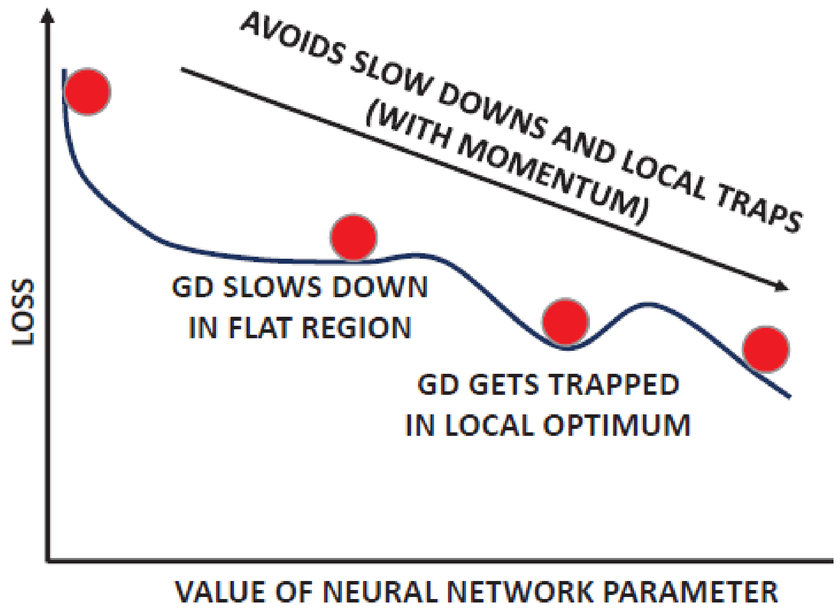
$\Rightarrow$

$$\Delta w_{ij} = \alpha \delta_i x_j$$
$$m = \Delta w_{ij} + \boldsymbol{\beta} m^{-}$$
$$w_{ij} \leftarrow w_{ij} + m$$
$$m^{-} = m$$

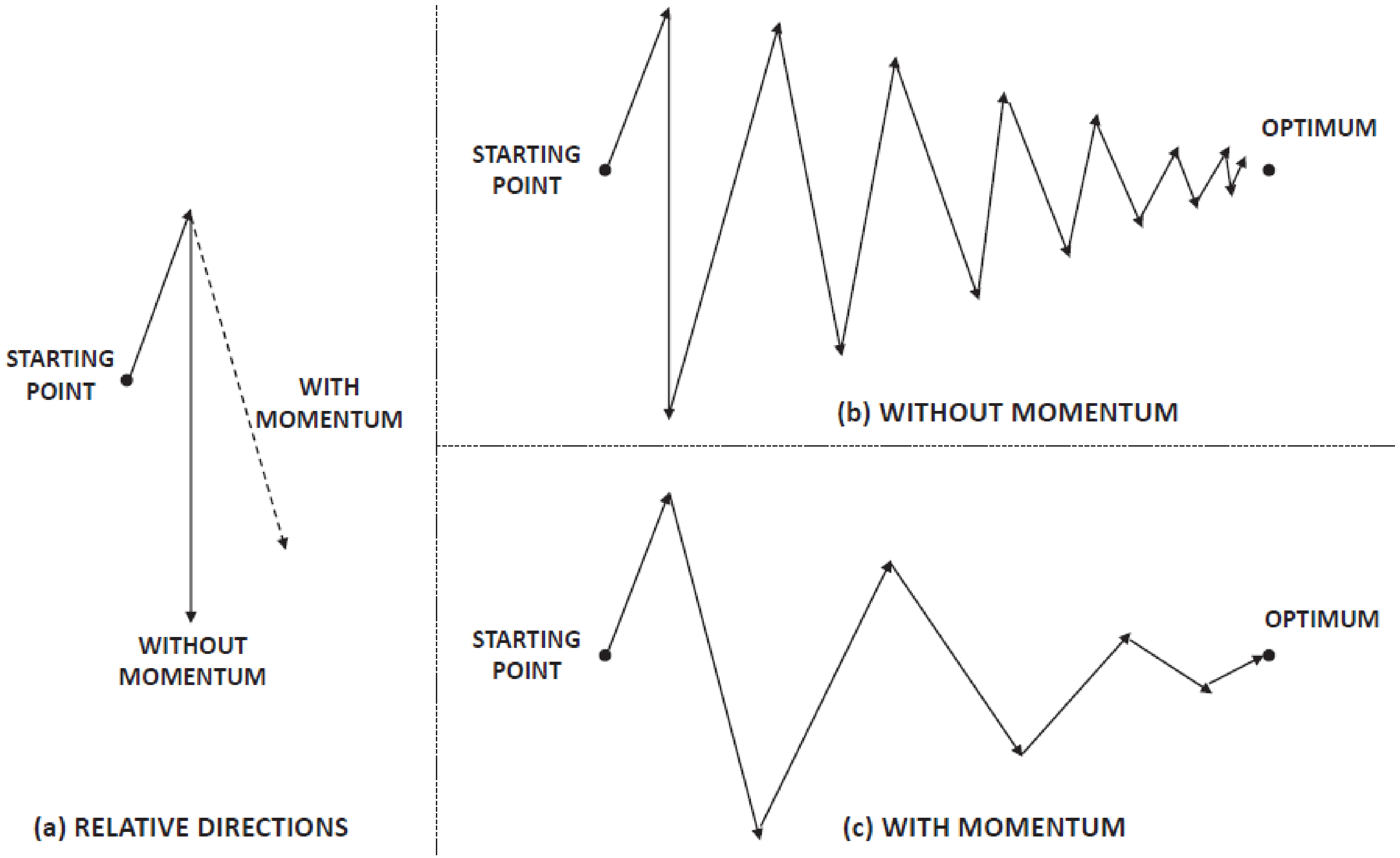
m^{-} is the previous momentum
β is a positive constant that is less than 1.

Momentum improves the learning stability.

Marble Rolling Down Hill



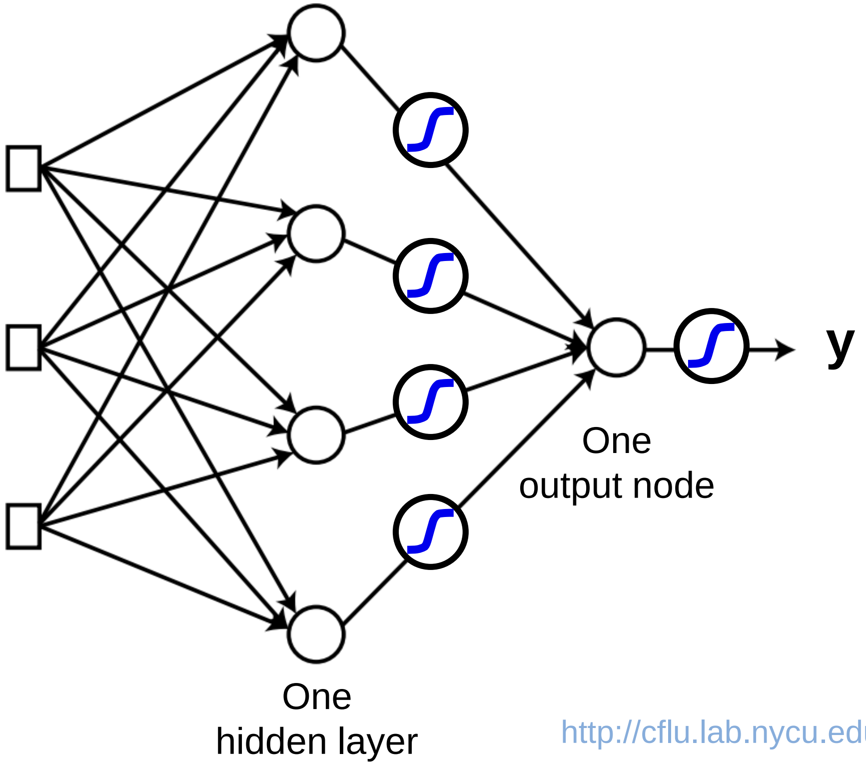
Avoiding Zig-Zagging with Momentum



Charu C. Aggarwal, Neural Networks and Deep Learning, Springer, 2018

Example: Back-Propagation with Momentum

- **MLmaterials_L10\Multi-layer**
  - **TestBackpropMmt.m**
  - **BackpropMmt.m**
  - **Sigmoid.m**



responses

{0, 0, 1, **0**}
{0, 1, 1, **1**}
{1, 0, 1, **1**}
{1, 1, 1, **0**}

with momentum

$$\begin{bmatrix} 0.0038 \\ 0.9929 \\ 0.9919 \\ 0.0127 \end{bmatrix}$$

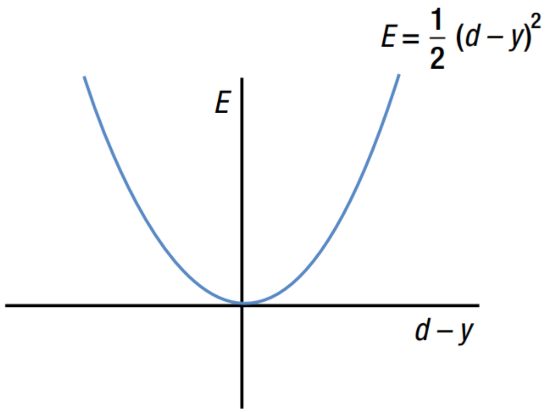
without momentum

$$\begin{bmatrix} 0.0060 \\ 0.9888 \\ 0.9891 \\ 0.0134 \end{bmatrix}$$

Cost Function and Learning Rule

- There are two primary types of cost functions

$$L = \sum_{i=1}^{M} \frac{1}{2}(d_i - y_i)^2$$



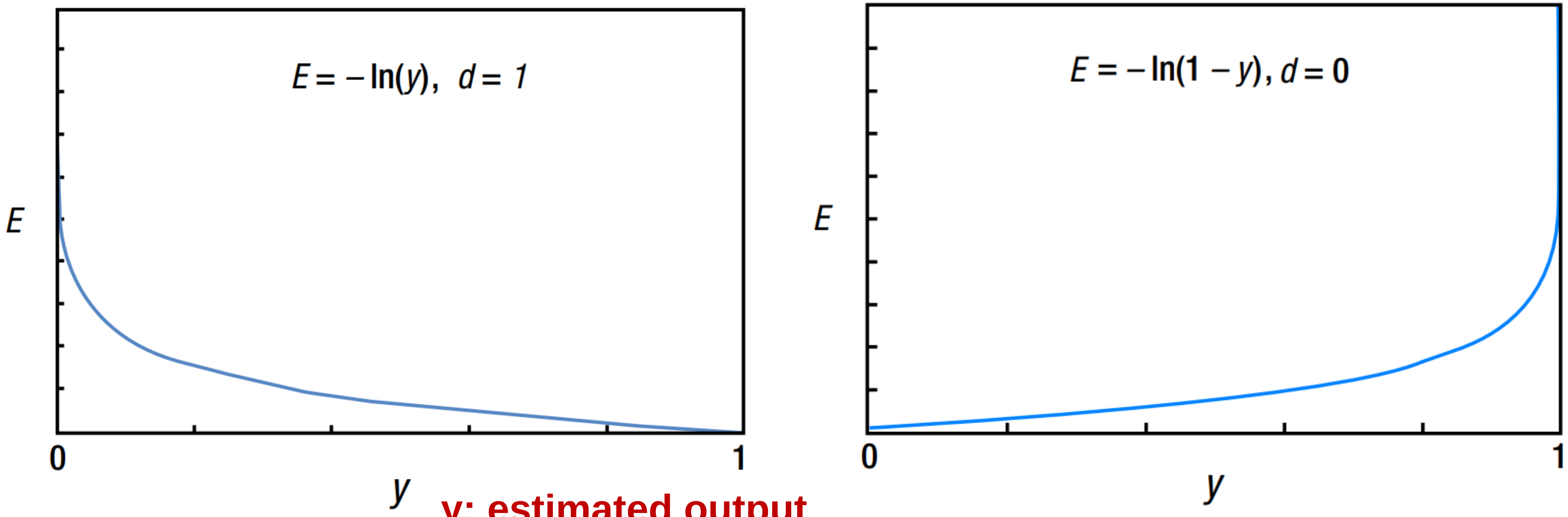
Quadratic function

$$L = \sum_{i=1}^{M} \{-d_i \ln(y_i) - (1 - d_i)\ln(1 - y_i)\}$$

Cross entropy function
(tend to have better performance)

Cross Entropy Function

d: correct output

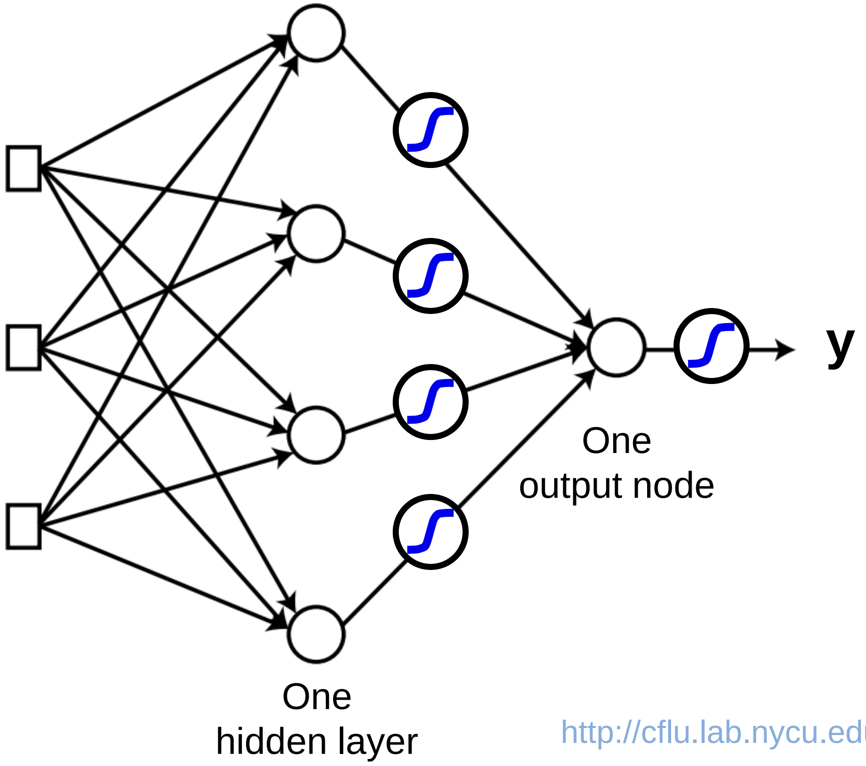


y: estimated output

- This cost function is proportional to the error.
- The cross entropy function is much more sensitive to the error than quadratic function.

Example:
Back-Propagation using cross entropy

- **MLmaterials_L10\Multi-layer**
  - **TestBackpropCE.m**
  - **BackpropCE.m**
  - **Sigmoid.m**



responses

{0, 0, 1, **0**}
{0, 1, 1, **1**}
{1, 0, 1, **1**}
{1, 1, 1, **0**}

Using cross entropy

$$\begin{bmatrix} 0.00003 \\ 0.9999 \\ 0.9998 \\ 0.00036 \end{bmatrix}$$

with momentum

$$\begin{bmatrix} 0.0038 \\ 0.9929 \\ 0.9919 \\ 0.0127 \end{bmatrix}$$

Regularization

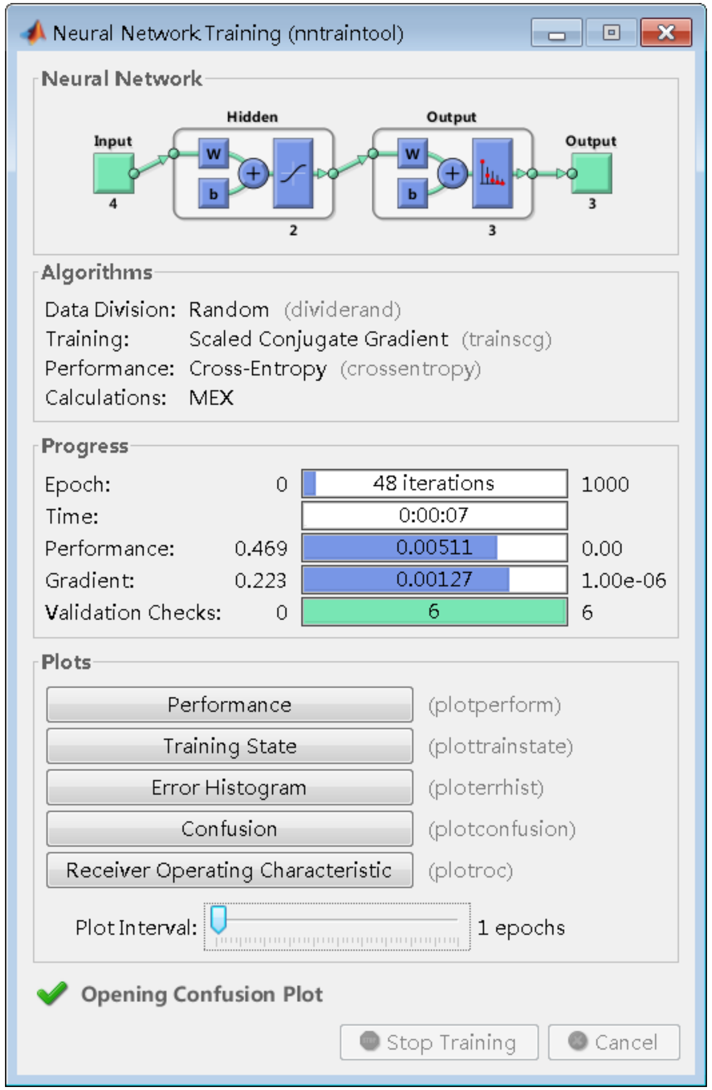
- One of the primary approaches used to overcome overfitting is making the model as simple as possible using regularization.

- In a mathematical sense, the essence of regularization is adding the sum of the weights to the cost function.

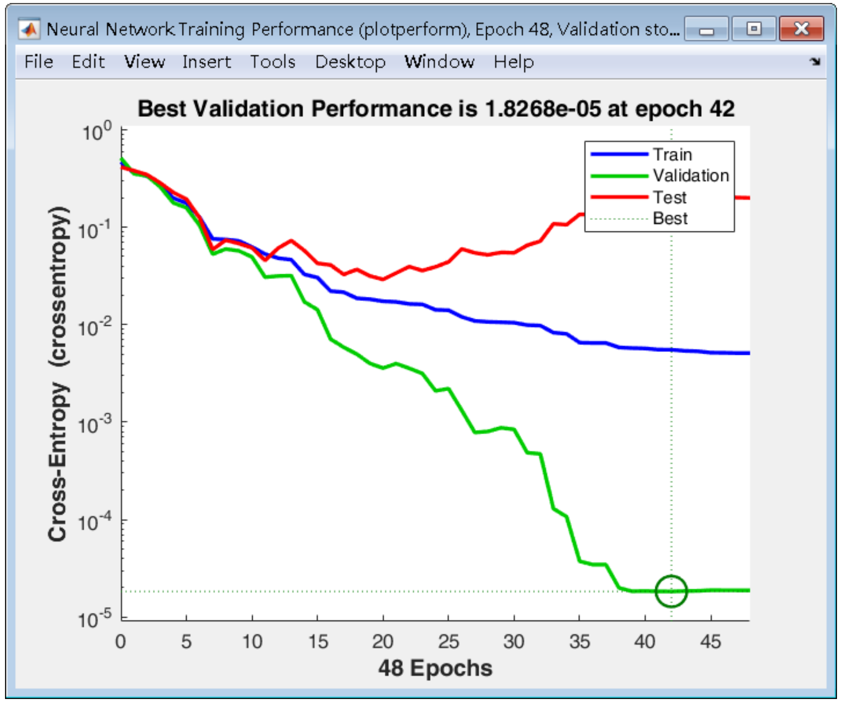
$$J = \frac{1}{2}\sum_{i=1}^{M}(d_i - y_i)^2 + \boldsymbol{\lambda}\frac{\mathbf{1}}{\mathbf{2}}\|\boldsymbol{w}\|^2$$

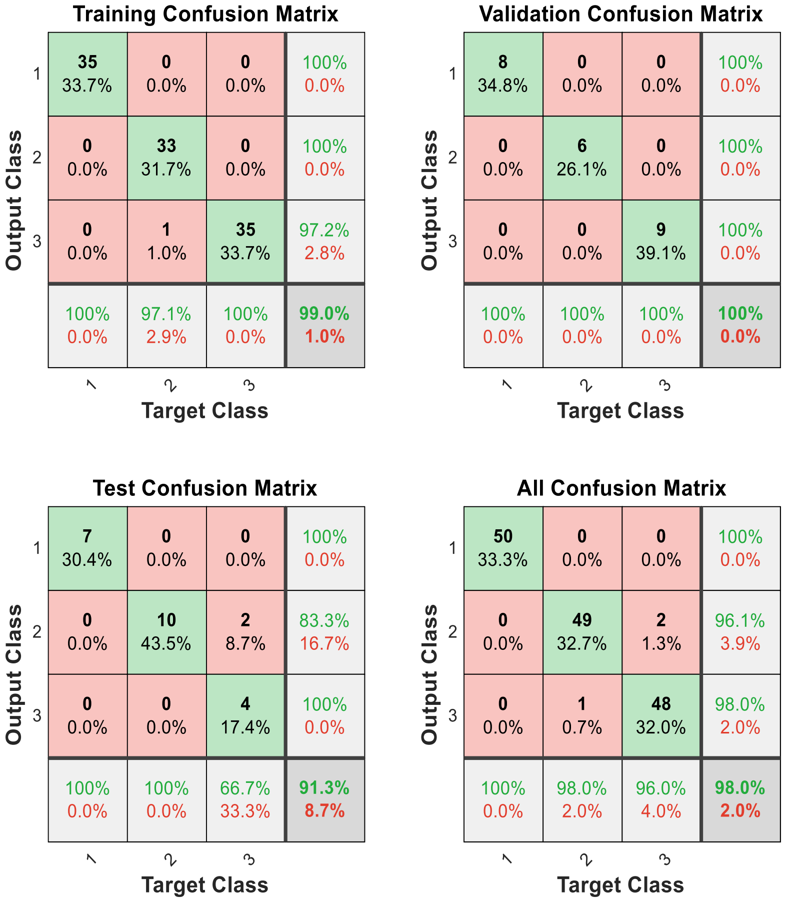
$$J = \sum_{i=1}^{M}\{-d_i\ln(y_i) - (1-d_i)\ln(1-y_i)\} + \boldsymbol{\lambda}\frac{\mathbf{1}}{\mathbf{2}}\|\boldsymbol{w}\|^2$$

nnstart



Iris dataset





THE END

Contact:

盧家鋒 alvin4016@nycu.edu.tw