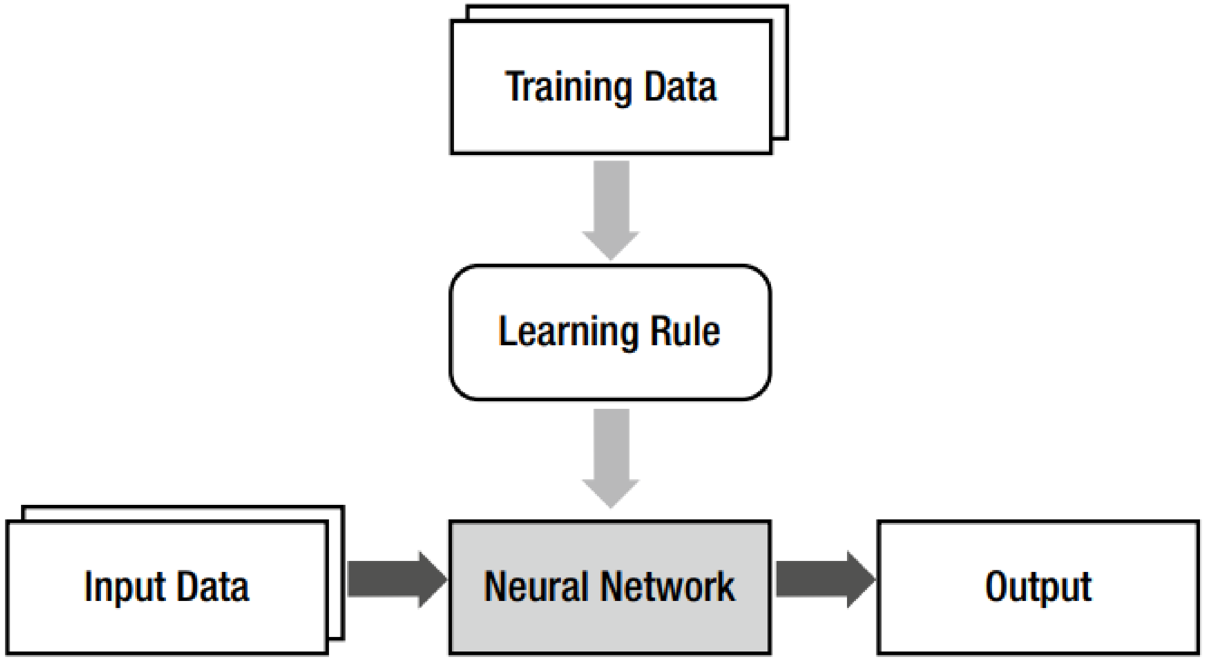
Neural Network

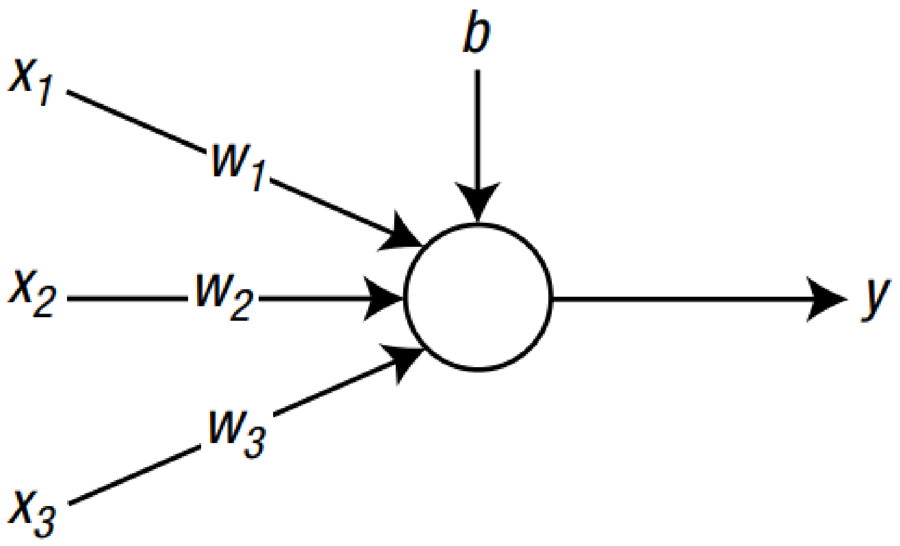
生醫光電所 吳育德

Phil Kim, MATLAB Deep Learning With Machine Learning, Neural Networks and Artificial Intelligence, 2017, chapter 2.

Neural Network

- The models of Machine Learning can be implemented in various forms. The neural network is one of them.





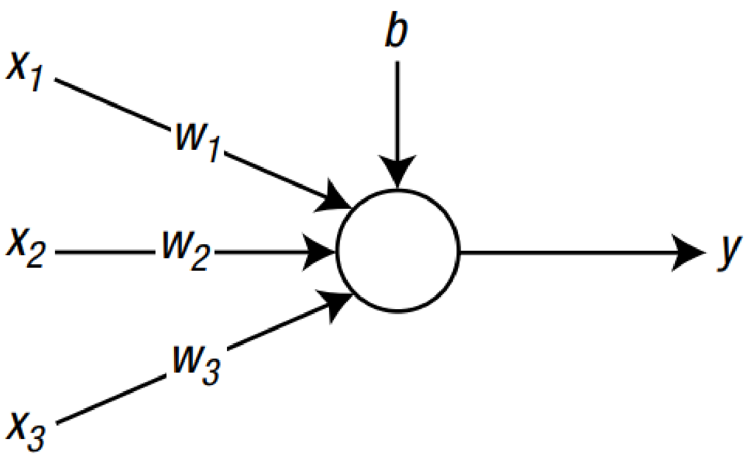
A node that receives three inputs.

x1, x2, and x3 are the **input signals.**

w1, w2, and w3 are the **weights** for the corresponding signals.

b is the **bias**.

- The circle and arrow of the figure denote the node and signal flow, respectively.
- The information of the neural net is stored in the form of weights and bias.



The equation of the weighted sum can be written with matrices as： $v = wx + b$ **(Equation 2.1)**

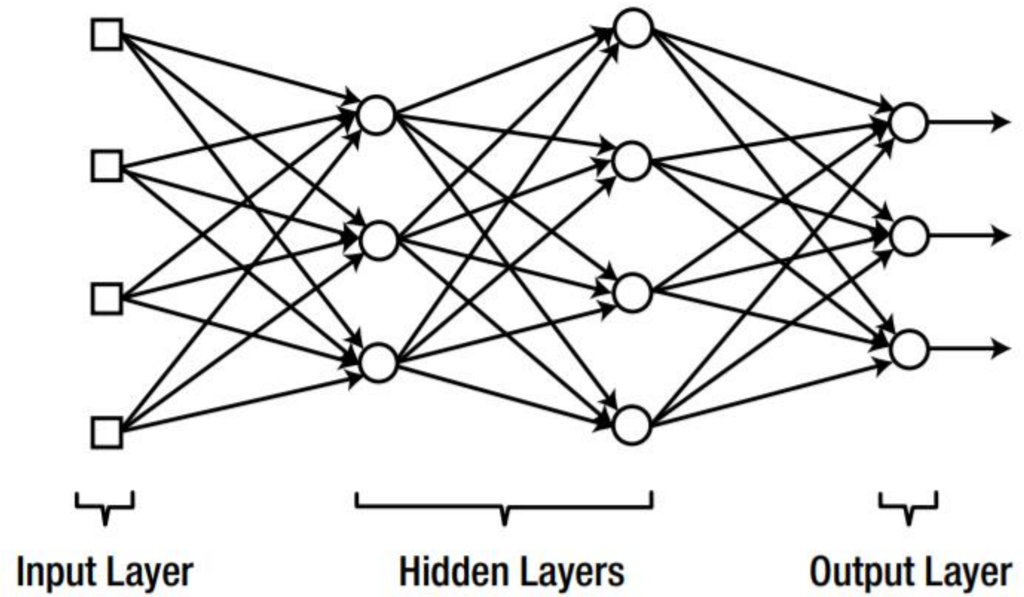
Where w and x are defined as： $w = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix}$ $x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$

Finally, the node enters the weighted sum into the activation function and yields its output：

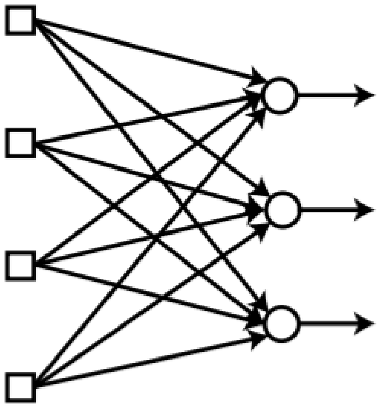
$$y = \varphi(v)$$

Layers of Neural Network

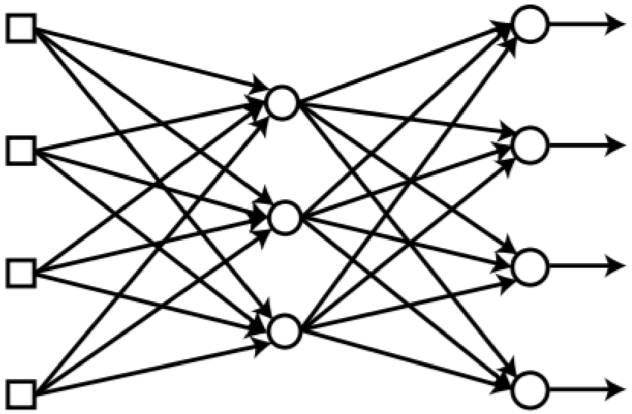
- The group of square nodes in figure is called the **input layer**. They do not calculate the weighted sum and activation function.
- The group of the rightmost nodes is called the **output layer**. The output from these nodes becomes the final result of the neural network.
- The layers in between the input and output layers are called **hidden layers**.



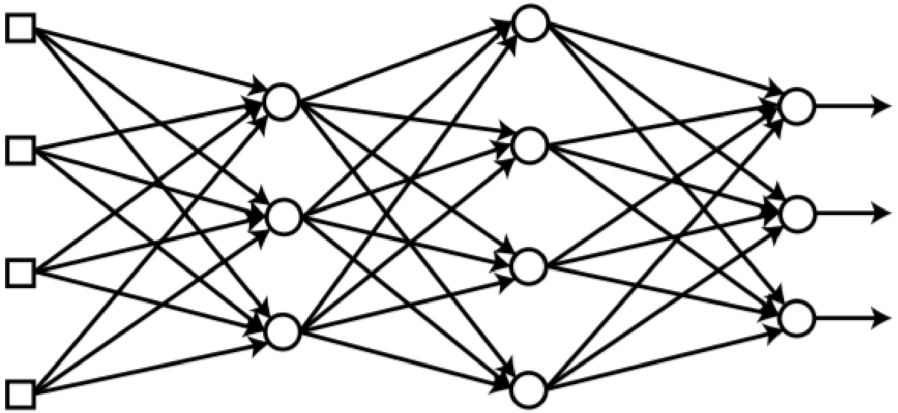
Single-Layer Neural Network		Input Layer – Output Layer
Multi-Layer Neural Network	Shallow Neural Network	Input Layer – Hidden Layer – Output Layer
	Deep Neural Network	Input Layer – Hidden Layers – Output Layers



Single-layer Neural Network

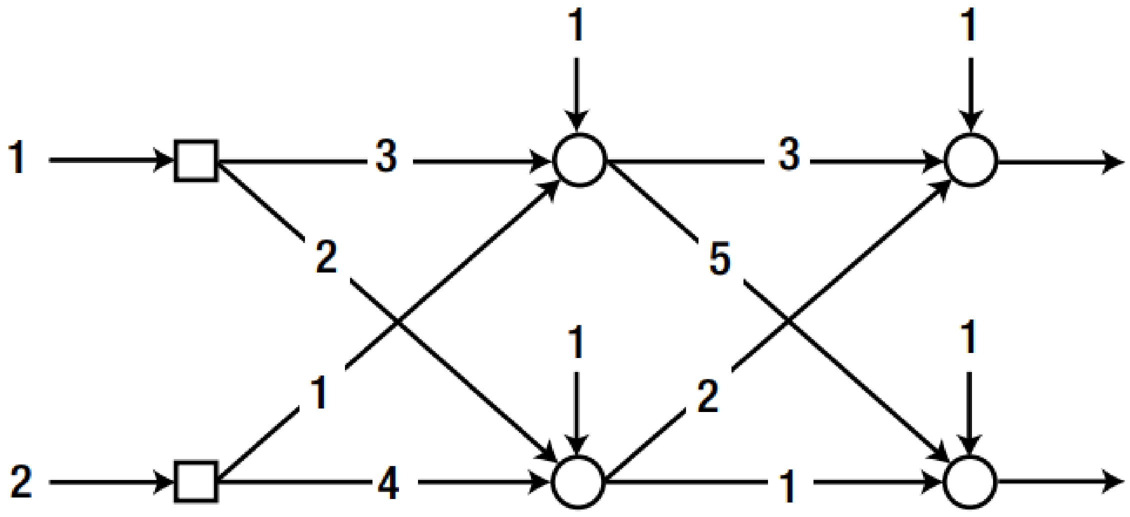


(Shallow) Multi-layer Neural Network

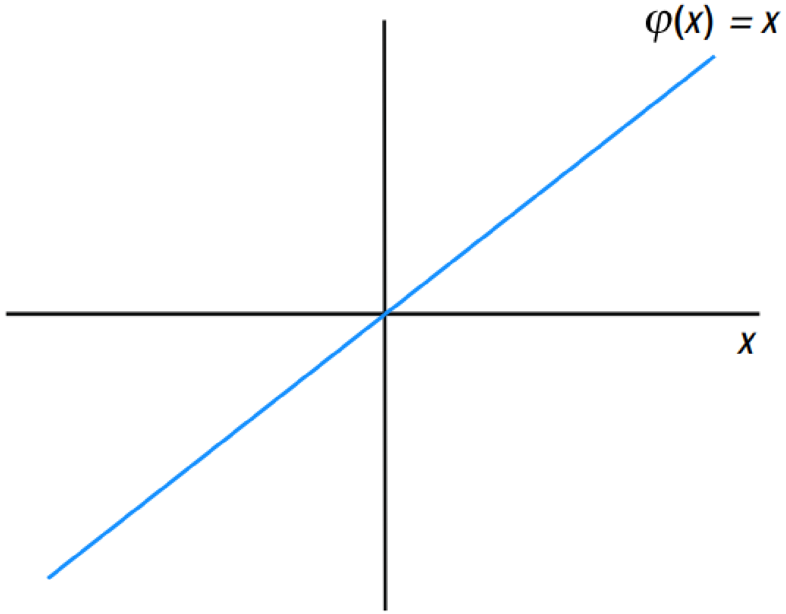


Deep Neural Network

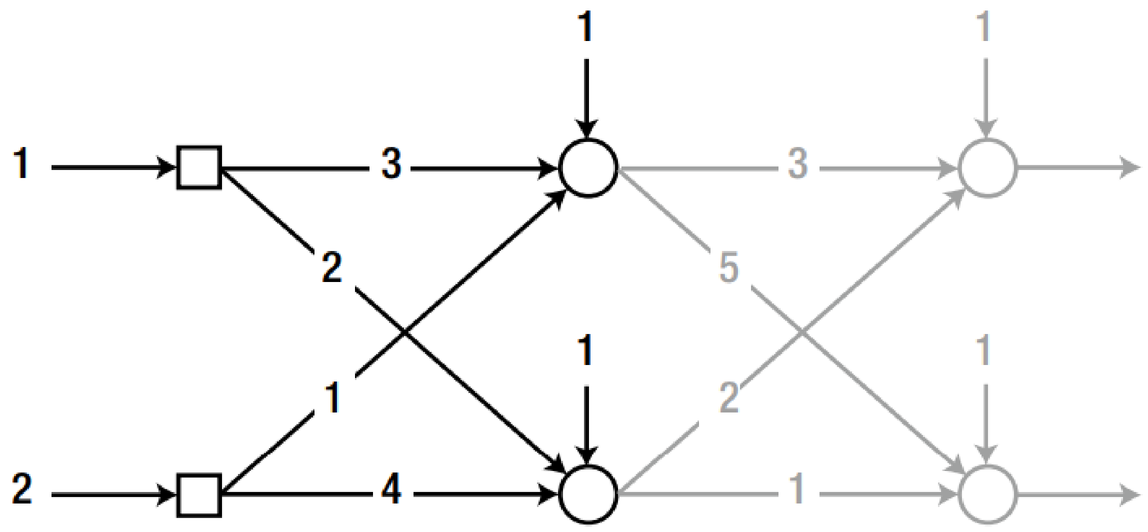
Example



A neural network with a single hidden layer.



The activation function of each node is a linear function.



The first node of the hidden layer calculates the output as:

Weighted sum: $v = (3 \times 1) + (1 \times 2) + 1 = 6$

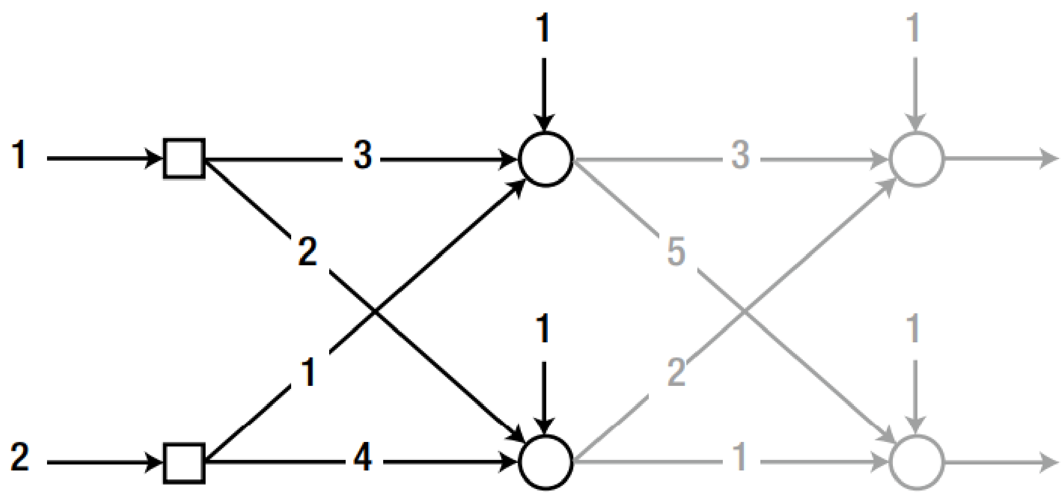
Output: $y = \varphi(v) = v = 6$

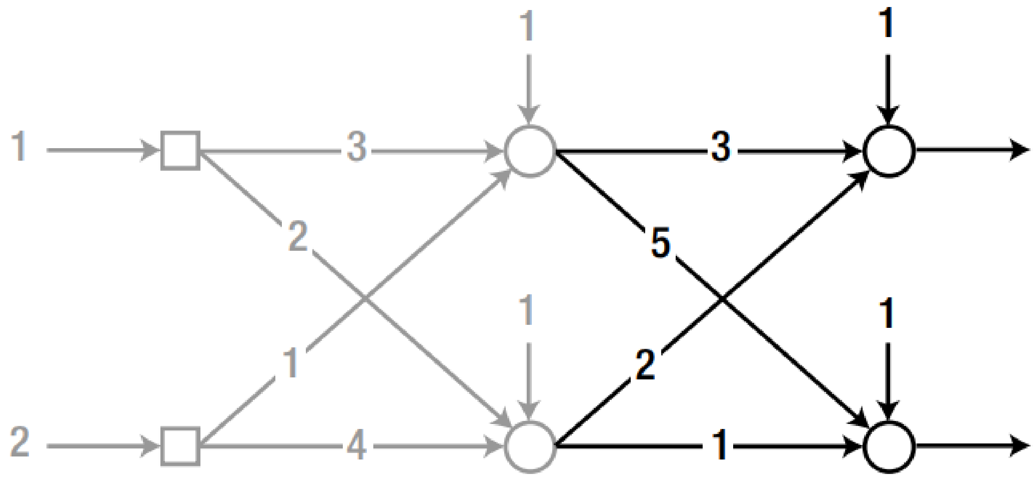
In a similar manner, the second node of the hidden layer calculates the output as:

Weighted sum: $v = (2 \times 1) + (4 \times 2) + 1 = 11$

Output: $y = \varphi(v) = v = 11$

Matrix equation





$$v = Wx + b$$

$$W = \begin{bmatrix} \text{-- weights of the first node --} \\ \text{-- weights of the second node --} \end{bmatrix} = \begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix}$$

$$v = \begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix}\begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 11 \end{bmatrix}$$

$$\text{Output: } y = \varphi(v) = v = \begin{bmatrix} 6 \\ 11 \end{bmatrix}$$

$$\text{Weighted sum: } v = \begin{bmatrix} 3 & 2 \\ 5 & 1 \end{bmatrix}\begin{bmatrix} 6 \\ 11 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 41 \\ 42 \end{bmatrix}$$

$$\text{Output: } y = \varphi(v) = v = \begin{bmatrix} 41 \\ 42 \end{bmatrix}$$

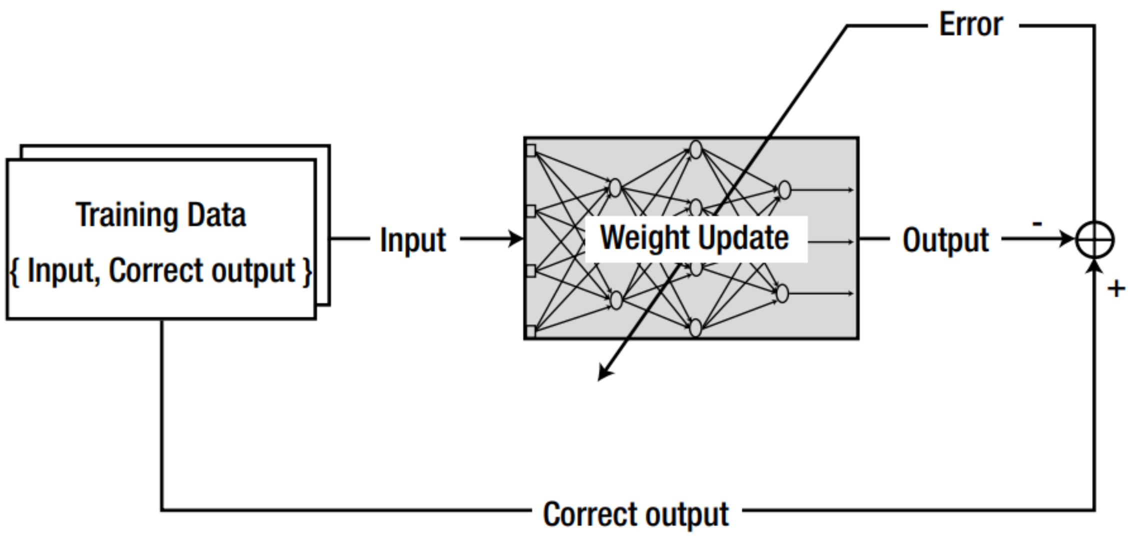
Supervised Learning of a Neural Network

1. Initialize the weights with adequate values.

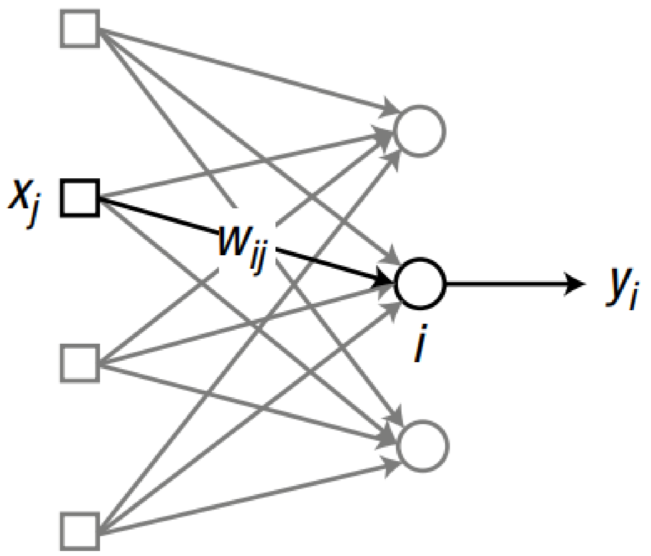
2. Take the “input” from the training data { input, correct output }, and enter it into the neural network. Obtain the output from the neural network and **calculate the error from the correct output**.

3. **Adjust the weights to reduce the error.**

4. Repeat Steps 2-3 for all training data



Training of a Single-Layer Neural Network: Delta Rule



$$e_i \triangleq d_i - y_i$$

d_i is the correct output of the output node i.

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

$$w_{ij} \leftarrow w_{ij} + \alpha e_i x_j$$ **(Equation 2.2)**

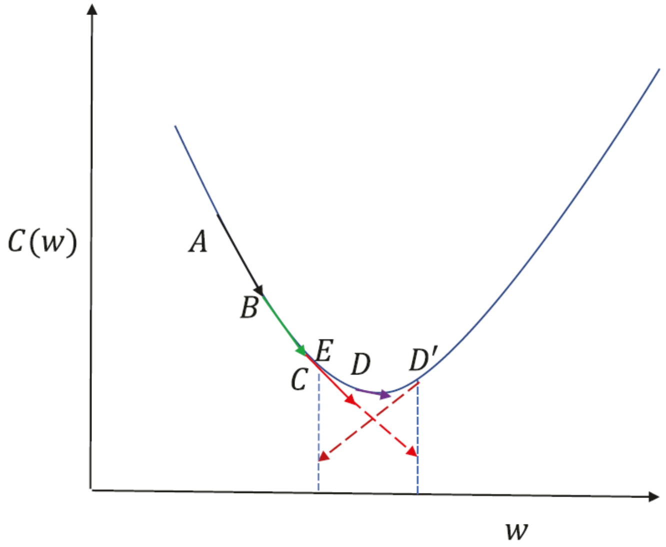
- Let us define the loss function for output node y_i

$$L = \frac{1}{2}(d_i - y_i)^2,\ \ e_i = d_i - y_i,\ \ y_i = \sum_{j=1}^{m} w_{ij} x_j$$

where m is the numbers of input nodes

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

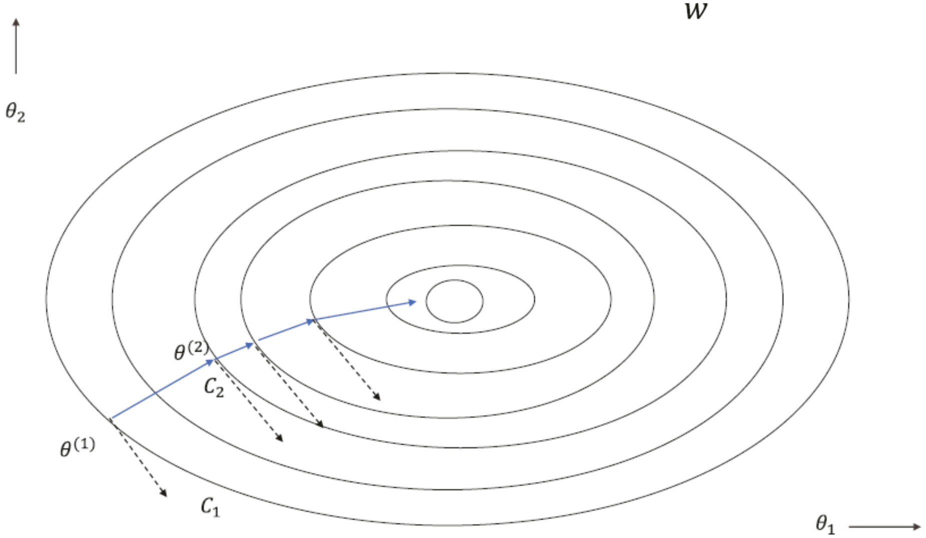
$$\frac{\partial L}{\partial w_{ij}} = e_i(-1)\frac{\partial y_i}{\partial w_{ij}} = -e_i x_j$$



- The steepest (gradient) decent method

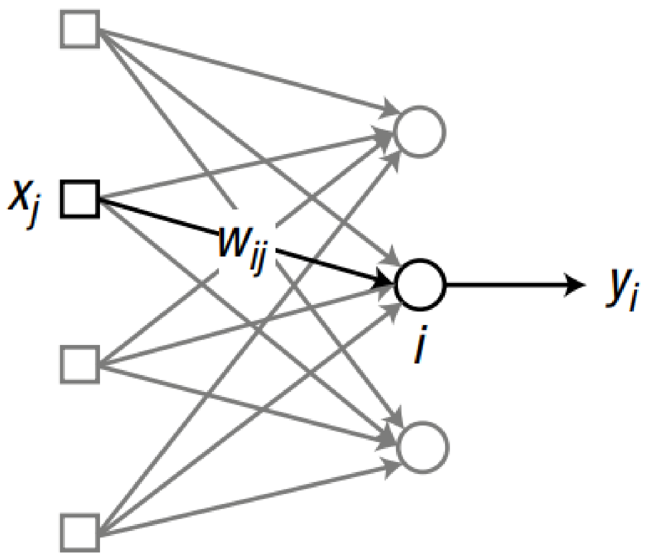
$$w_{ij}^{(k+1)} = w_{ij}^{(k)} - \alpha \frac{\partial L}{\partial w_{ij}}$$

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



- Or express as

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



$$e_i \triangleq d_i - y_i$$

$$\boxed{w_{ij} \leftarrow w_{ij} + \alpha e_i x_j}$$ **(Equation 2.2)**

x_j = The input node *j*, (j =1, 2, 3, 4)
e_i = The error of the output node *i*
w_{ij} = The weight between the output node i and input node j
α = Learning rate ($0 < \alpha \leq 1$)

The learning rate,α , determines how much the weight is changed per time.
If this value is too high, the output wanders around the solution and fails to converge.
In contrast, if it is too low, the calculation reaches the solution too slowly.

1. Initialize the weights at adequate values.

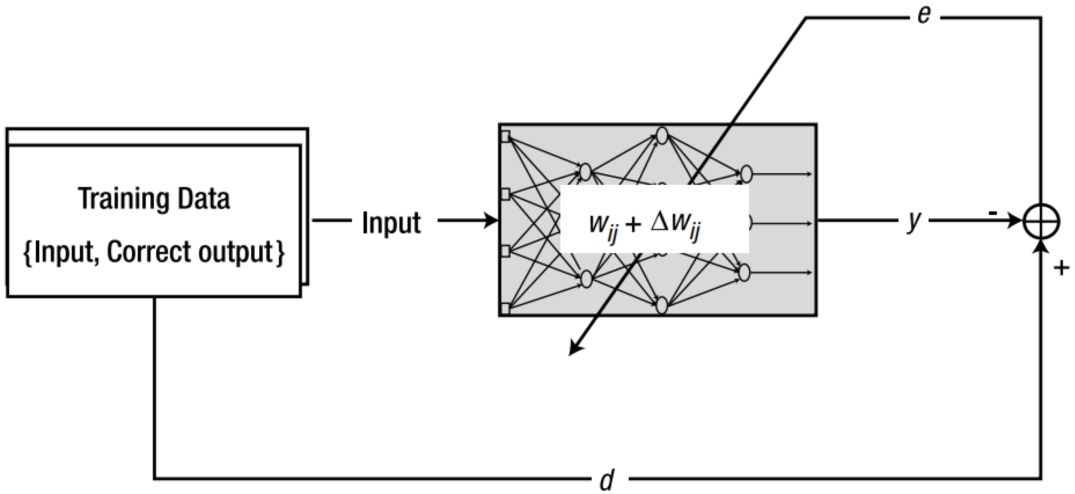
2. Take the "input" from the training data of { input, correct output } and enter it to the neural network. Calculate the error of the output, y_i, from the correct output, d_i, to the input.

$$e_i = d_i - y_i$$

3. Calculate the delta rule:

$$\Delta w_{ij} = \alpha e_i x_j$$

4. Adjust the weights as: $w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$



5. Perform Steps 2~4 for all training data.

6. Repeat Steps 2~5 until the error reaches an acceptable tolerance level.
(All training data goes through Steps 2-5 once, is called an epoch.)

Generalized Delta Rule

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

$$w_{ij} \leftarrow w_{ij} + \alpha \delta_i x_j$$ **(Equation 2.3)**

- It is the same as the delta rule of the previous section, except that e_i is replaced with δ_i

$$\delta_i = \varphi'(v_i) e_i$$ **(Equation 2.4)**

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*

- Let us define the loss function for output node y_i

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

where m is the numbers of input nodes

- 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 decent method

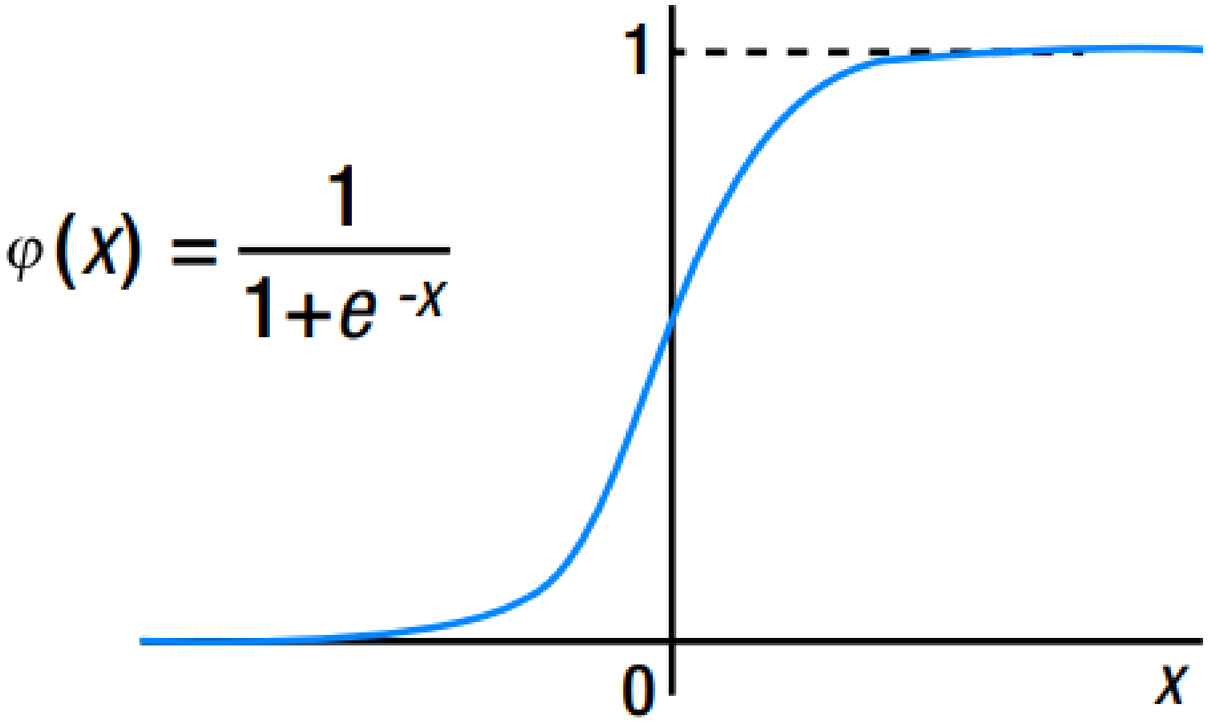
$$w_{ij}^{(k+1)} = w_{ij}^{(k)} - \alpha \frac{\partial L}{\partial w_{ij}}$$

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

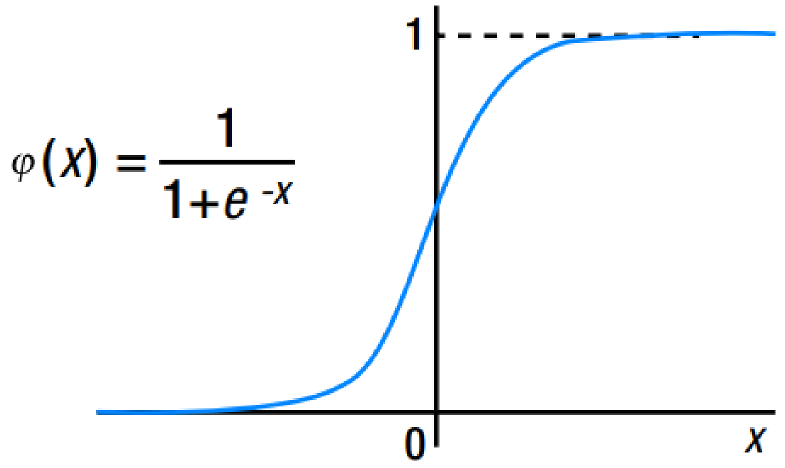
- Or we may express the above equation as

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

- We can derive the delta rule with the **sigmoid function**, which is widely used as an activation function



The sigmoid function



The sigmoid function

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

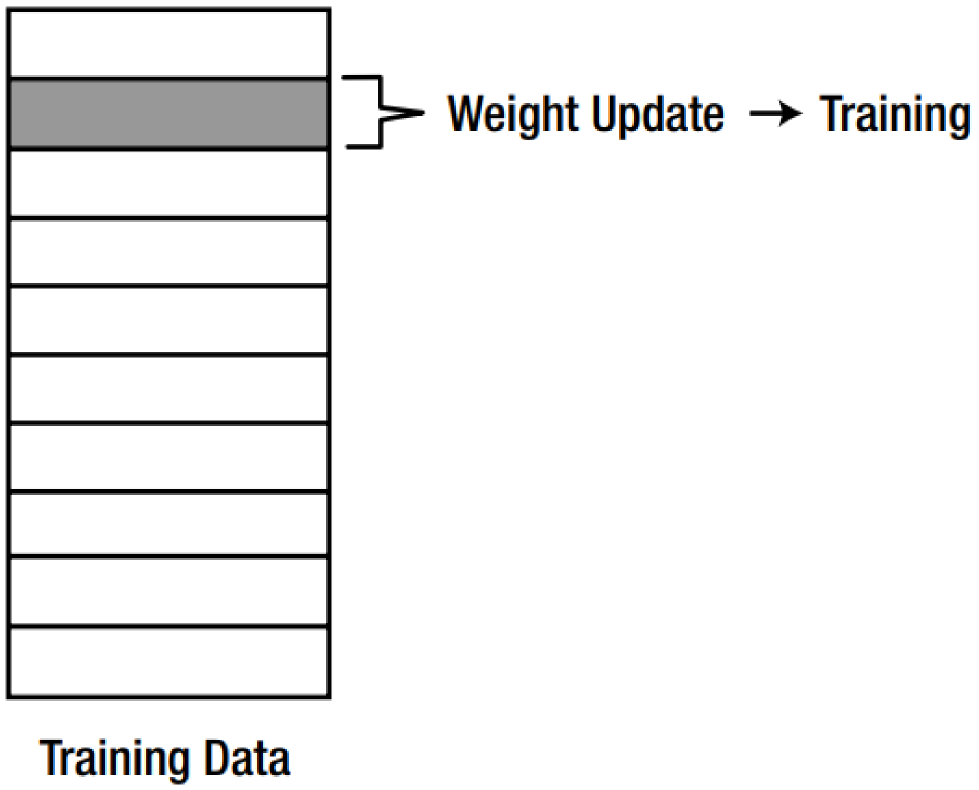
$$\varphi'(x) = \varphi(x)\big(1-\varphi(x)\big)$$

$$\delta_i = \varphi'(v_i)e_i \quad \Rightarrow \quad \delta_i = \varphi'(v_i)e_i = \varphi(v_i)\big(1-\varphi(v_i)\big)e_i$$

$$w_{ij} \leftarrow w_{ij} + \alpha\,\varphi(v_i)\big(1-\varphi(v_i)\big)e_i x_j$$ **(Equation 2.5)**

Stochastic Gradient Descent

- The Stochastic Gradient Descent (SGD) calculates the error for each training data and adjusts the weights immediately.
- If we have 100 training data points, the SGD adjusts the weights 100 times.



The SGD calculates the weight updates as: $\Delta w_{ij} = \alpha \delta_i x_j$

Batch

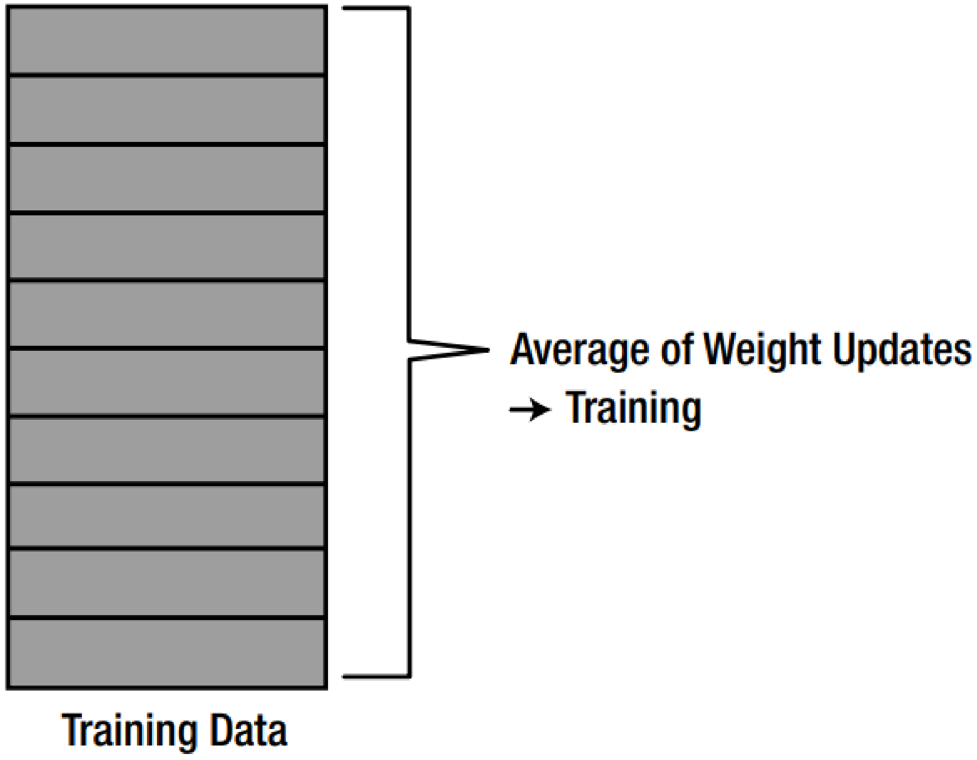
- In the batch method, each weight update is calculated for **all errors of the training data**, and the **average of the weight updates** is used for adjusting the weights.

The batch method calculates the weight update as:

$$\Delta w_{ij} = \frac{1}{N}\sum_{k=1}^{N}\Delta w_{ij}(k)$$

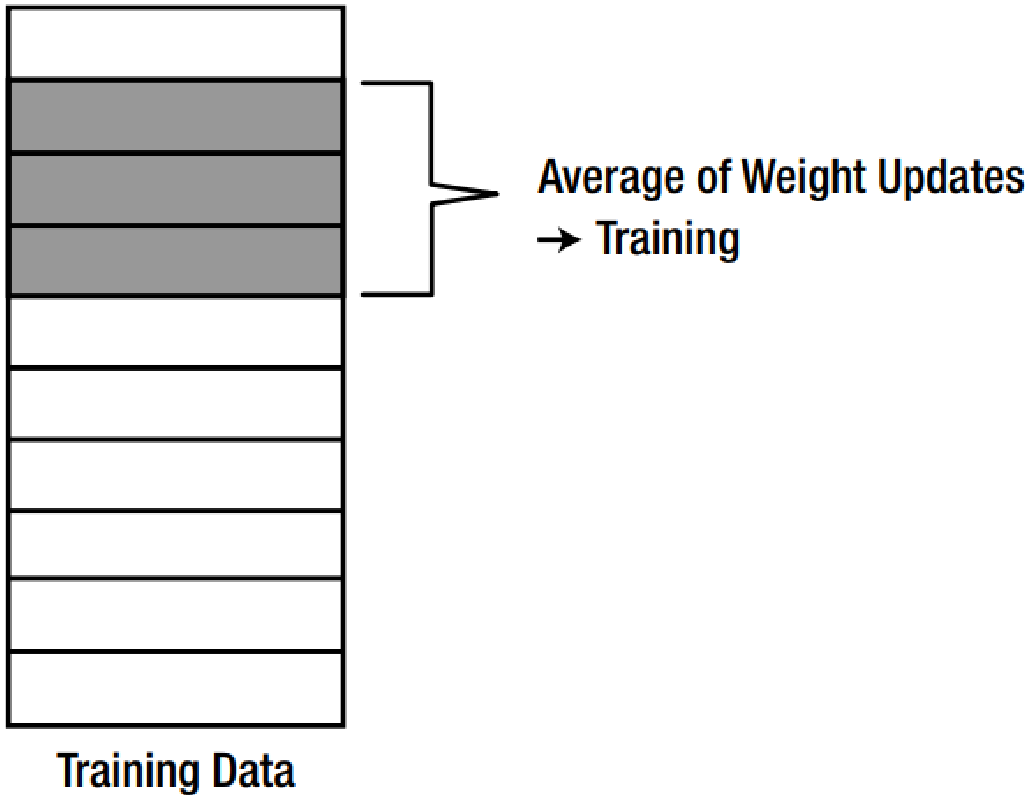
(Equation 2.6)

$\Delta w_{ij}(k)$ is the weight update for the k-th training data and N is the total number of the training data.



Mini Batch

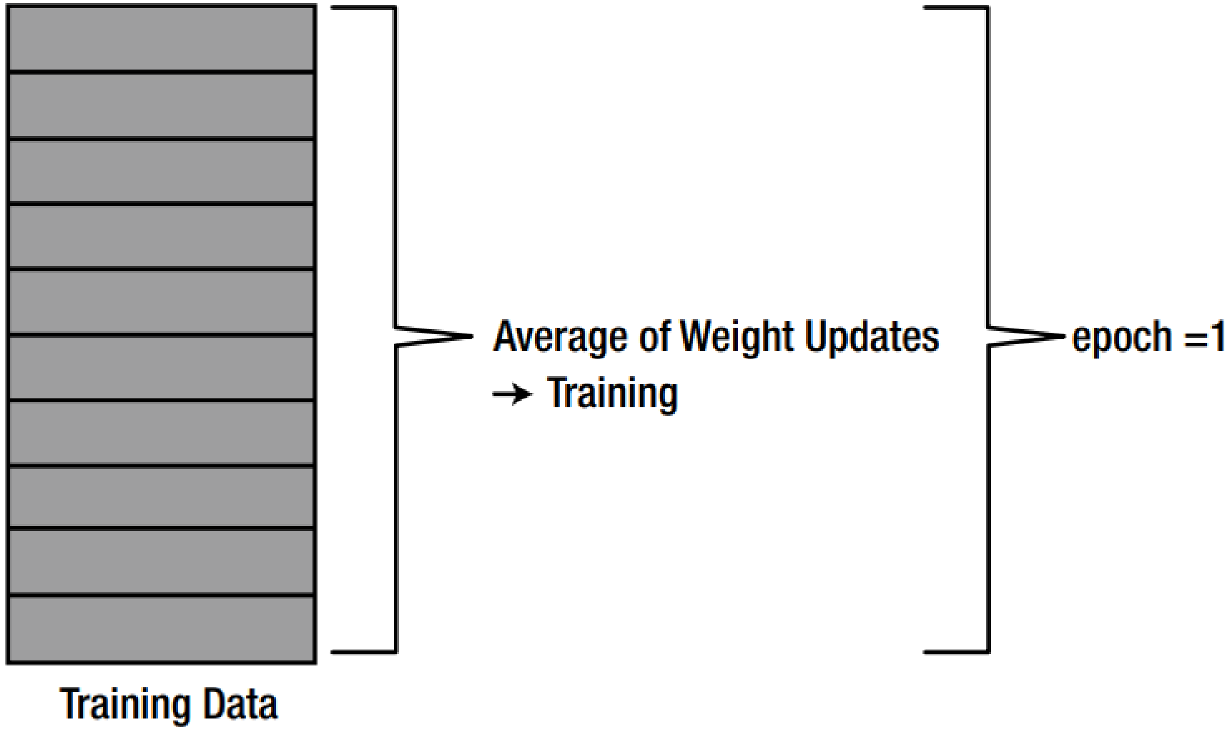
- It selects **a part of the training dataset** and uses them for training in the batch method.
- It calculates the weight updates of the selected data and trains the neural network with the averaged weight update.
- It is often utilized in Deep Learning, which manipulates a **significant amount of data**.
- Have speed from the SGD and stability from the batch.



- The epoch is the **number of completed training cycles for all of the training data**.

- In the batch method, the number of training cycles of the neural network equals an epoch.

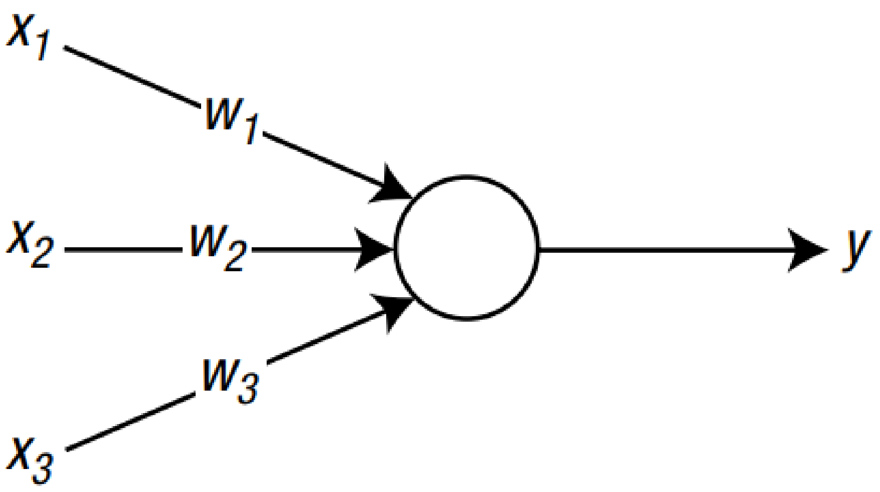
- In the mini batch, the number of training processes for one epoch varies depending on the number of data points in each batch.

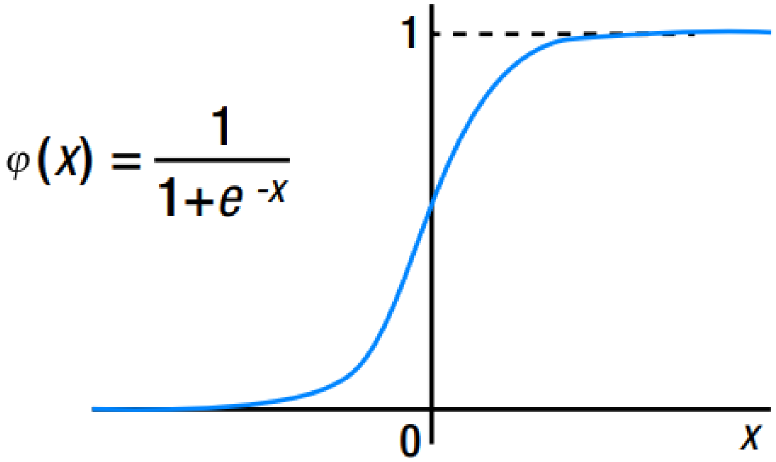


Example: Delta Rule

- Consider a neural network that consists of three input nodes and one output node.

- The sigmoid function is used for the activation function of the output node.





The sigmoid function defined

- We have four training data points.

- As they are used for supervised learning, each data point consists of an input-correct output pair.

- The last bold number of each dataset is the correct output.

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

- The delta rule for the sigmoid function, which is given by Equation 2.5, is the learning rule.

$$w_{ij} \leftarrow w_{ij} + \alpha \underline{\varphi(v_i)(1-\varphi(v_i))e_i} x_j \qquad \textbf{(Equation 2.5)}$$

- Equation 2.5 can be rearranged as a step-by-step process, as follows:

$$\begin{aligned} \delta_i &= \varphi(v_i)(1-\varphi(v_i))e_i \\ \Delta w_{ij} &= \alpha \delta_i x_j \\ w_{ij} &\leftarrow w_{ij} + \Delta w_{ij} \end{aligned} \qquad \textbf{(Equation 2.7)}$$

Implementation of the SGD Method

- The function **DeltaSGD** is the SGD method of the delta rule given by Equation 2.7.

$$\begin{aligned} \delta_i &= \varphi(v_i)\left(1-\varphi(v_i)\right)e_i \\ \Delta w_{ij} &= \alpha\,\delta_i\,x_j \\ w_{ij} &\leftarrow w_{ij} + \Delta w_{ij} \end{aligned}$$

(Equation 2.7)

Coding

Algorithm implementation	example/ DeltaSGD.m
Test program	example/ TestDeltaSGD.m

- Take one of the data points and **calculate the output**, y.
- **Calculate the difference** between this output and the correct output, d.
- **Calculate the weight update**, dW, according to the delta rule.
- Using this weight update, **adjust the weight** of neural network.
- Repeat the process for the number of the training data points, N.

The function **DeltaSGD(W, X, D)**

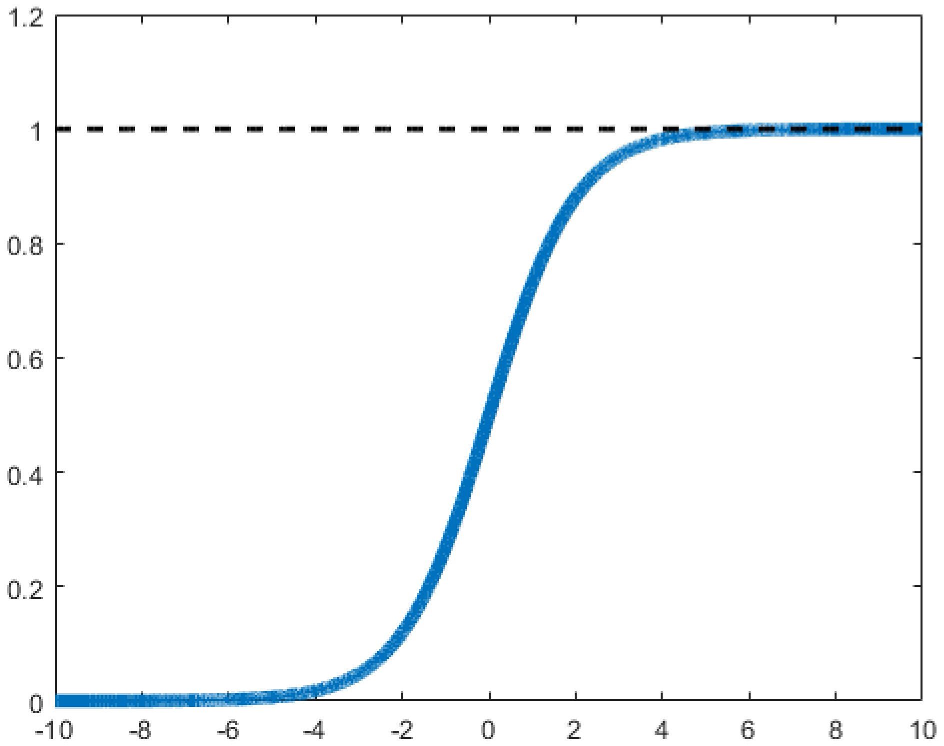
- **W** is the argument that carries the weights.
- **X** and **D** carry the inputs and correct outputs of the training data, respectively.

DeltaSGD.m

```
function W = DeltaSGD(W, X, D)
  alpha = 0.9;

  N = 4;
  for k = 1:N
    x = X(k, :)';
    d = D(k);

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

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

    dW = alpha*delta*x;     % delta rule

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

The function Sigmoid(x)

```
function y = Sigmoid(x)
  y = 1 / (1 + exp(-x));
end
```



TestDeltaSGD.m

- This program calls the function DeltaSGD, trains it 10,000 times, and displays the output from the trained neural network with the input of all the training data.

```
clear all

X = [ 0 0 1;
      0 1 1;
      1 0 1;
      1 1 1;
    ];

D = [ 0; 0; 1; 1];

W = 2*rand(1, 3) - 1;

for epoch = 1:10000           % train
  W = DeltaSGD(W, X, D);
end

N = 4;                        % inference
for k = 1:N
  x = X(k, :)';
  v = W*x;
  y = Sigmoid(v)
end
```

- This code initializes the weights with random real numbers between -1 and 1.

- Executing this code produces the following values. These output values are very close to the correct outputs in D.

$$\begin{bmatrix} 0.0102 \\ 0.0083 \\ 0.9932 \\ 0.9917 \end{bmatrix} \Leftrightarrow \qquad D = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

Implementation of the Batch Method

- DeltaBatch.m does not immediately train the neural network with the weight update, dW, of the individual training data points.
- It adds the individual weight updates of the entire training data to dWsum and adjusts the weight just once using the average, dWavg.
- This is the fundamental difference that separates this method from the SGD method.

```
function W = DeltaBatch(W, X, D)
  alpha = 0.9;

  dWsum = zeros(3, 1);

  N = 4;
  for k = 1:N
    x = X(k, :)';
    d = D(k);

    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
```

$$\Delta w_{ij} = \frac{1}{N}\sum_{k=1}^{N}\Delta w_{ij}(k)$$

DeltaSGD.m

```matlab
function W = DeltaSGD(W, X, D)
  alpha = 0.9;

  N = 4;
  for k = 1:N
    x = X(k, :)';
    d = D(k);

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

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

    dW = alpha*delta*x;      % delta rule

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

```matlab
function W = DeltaBatch(W, X, D)
  alpha = 0.9;

  dWsum = zeros(3, 1);

  N = 4;
  for k = 1:N
    x = X(k, :)';
    d = D(k);

    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
```

- TestDeltaBatch.m calls in the function DeltaBatch and trains the neural network 40,000 times.

- All the training data is fed into the trained neural network, and the output is displayed.

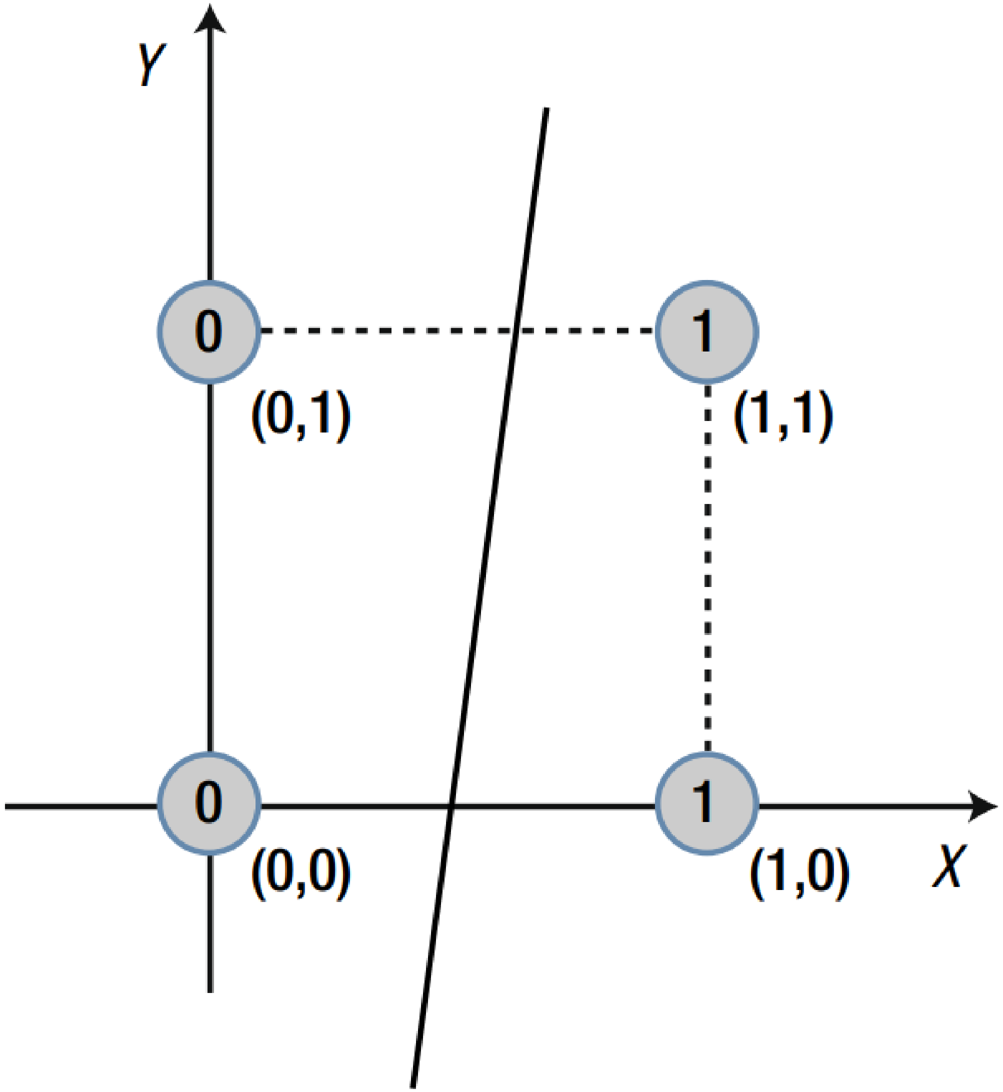
$$\begin{bmatrix} 0.0102 \\ 0.0083 \\ 0.9932 \\ 0.9917 \end{bmatrix} \Leftrightarrow \quad D = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

```
clear all

X = [ 0 0 1;
      0 1 1;
      1 0 1;
      1 1 1;
    ];

D = [ 0
      0
      1
      1
    ];

W = 2*rand(1, 3) - 1;

for epoch = 1:40000
  W = DeltaBatch(W, X, D);
end

N = 4;
for k = 1:N
  x = X(k, :)';
  v = W*x;
  y = Sigmoid(v)
end
```

- As the third value, i.e. the Z coordinate, is fixed as 1, the training data can be visualized on a plane as shown in figure.

- In this case, a straight border line that divides the regions of 0 and 1 can be found easily.

- This is a linearly separable problem.

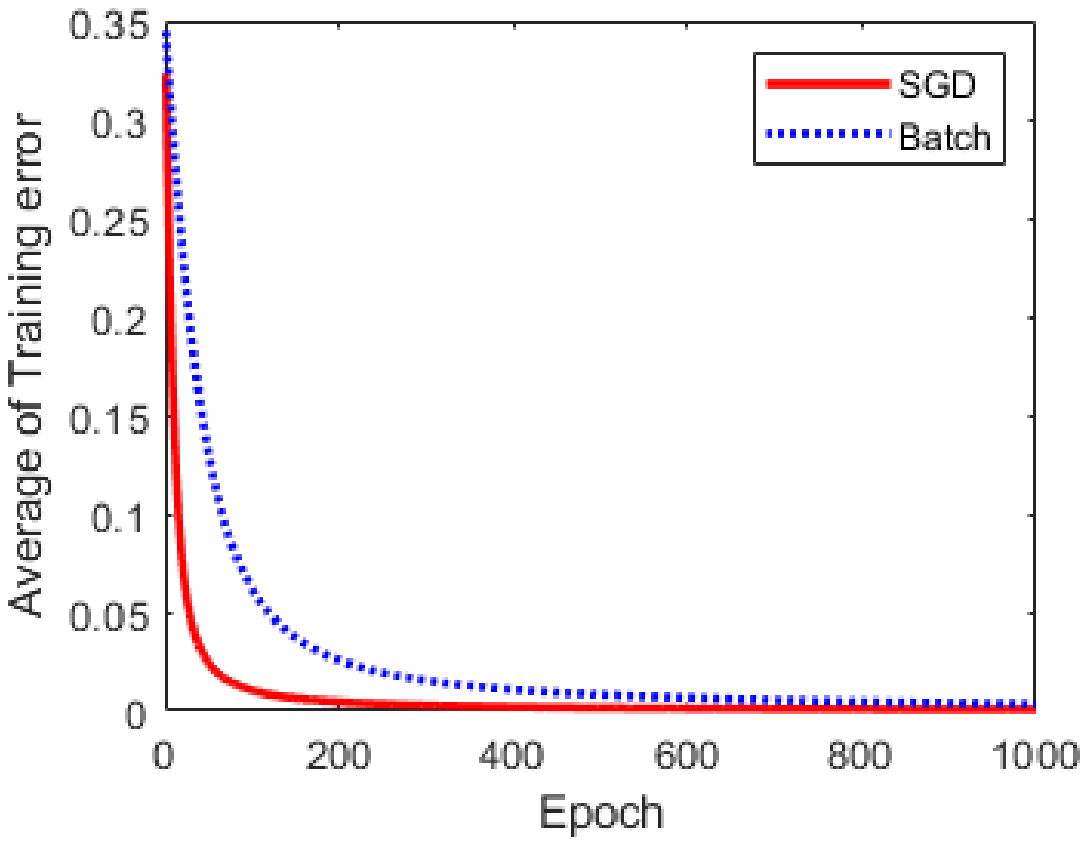


Comparison of the SGD and the Batch

- “SGDvsBatch.m” trains the neural network 1,000 times for each function, DeltaSGD and DeltaBatch.

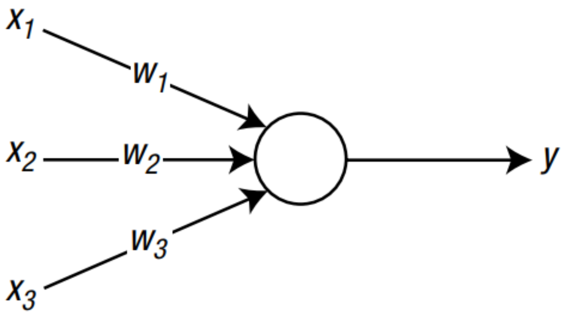
- At each epoch, it inputs the training data into the neural network and calculates the mean square error of the output.

- SGD yields faster reduction of the learning error than the batch; the SGD learns faster.



Limitations of Single-Layer Neural Networks

- Consider the same neural network that was discussed in the previous section.



- Assume that we have another four training data points, as shown in the table. It shouldn't cause any trouble, right?

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

- We will now train it with the delta rule using the SGD.
- When we run the code, the screen will show the following values, which consist of the output from the trained neural network corresponding to the training data.
- We can compare them with the correct outputs given by D.

$$\begin{bmatrix} 0.5297 \\ 0.5000 \\ 0.4703 \\ 0.4409 \end{bmatrix} \Leftrightarrow \quad D = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

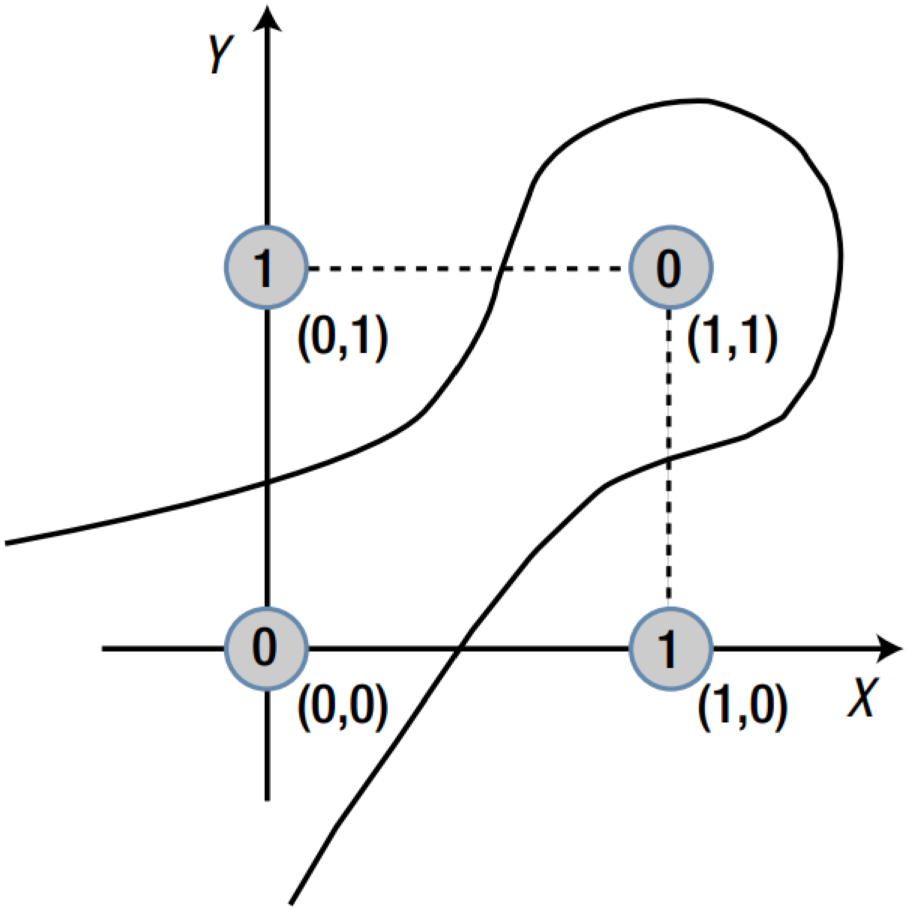
What happened?

```
clear all

X = [ 0 0 1;
      0 1 1;
      1 0 1;
      1 1 1;
    ];

D = [ 0
      1
      1
      0
    ];

W = 2*rand(1, 3) - 1;

for epoch = 1:40000          % train
  W = DeltaXOR(W, X, D);
end

N = 4;                       % inference
for k = 1:N
  x = X(k, :)';
  v = W*x;
  y = Sigmoid(v)
end
```

- One thing to notice from this figure is that **we cannot divide the regions of 0 and 1 with a straight line.**

- However, we may divide it with a complicated curve, as shown in figure.

- This type of problem is said to be **linearly inseparable**.



- 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.

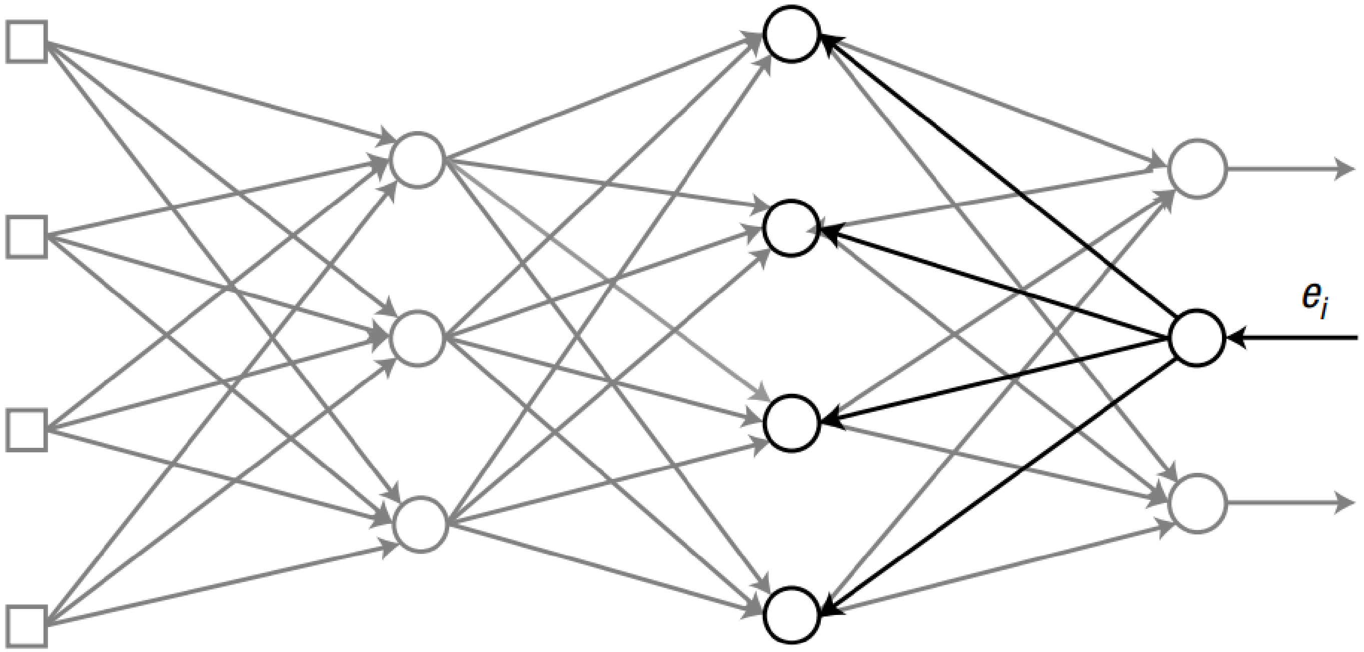
Training of Multi-Layer Neural Network

Phil Kim, MATLAB Deep Learning With Machine Learning, Neural Networks and Artificial Intelligence, 2017, chapter 3.

- In an effort to overcome the practical limitations of the single-layer, the neural network evolved into a multi-layer architecture.

- 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.



- In the back-propagation algorithm, the output error starts from the output layer and **moves backward** until it reaches the left next hidden layer to the input layer.

- In back-propagation, the signal still flows through the connecting lines and the weights are multiplied.

Back-Propagation Algorithm

- Consider a univariate logistic least-square model $L = \frac{1}{2}(\varphi(wx + b) - t)^2$

$$\frac{\partial L}{\partial w} = \frac{1}{2}\frac{\partial}{\partial w}(\varphi(wx + b) - t)^2$$
$$= (\varphi(wx + b) - t) \times \frac{\partial}{\partial w}(\varphi(wx + b) - t)$$
$$= (\varphi(wx + b) - t) \times \varphi'(wx + b) \times \frac{\partial}{\partial w}(wx + b)$$
$$= (\varphi(wx + b) - t) \times \varphi'(wx + b) \times x$$

$$\frac{\partial L}{\partial b} = \frac{1}{2}\frac{\partial}{\partial b}(\varphi(wx + b) - t)^2$$
$$= (\varphi(wx + b) - t) \times \frac{\partial}{\partial b}(\varphi(wx + b) - t)$$
$$= (\varphi(wx + b) - t) \times \varphi'(wx + b) \times \frac{\partial}{\partial b}(wx + b)$$
$$= (\varphi(wx + b) - t) \times \varphi'(wx + b)$$

- Disadvantages: Cumbersome calculation
  Two derivations are nearly identical (redundant)
  Repeated terms

Back-Propagation Algorithm

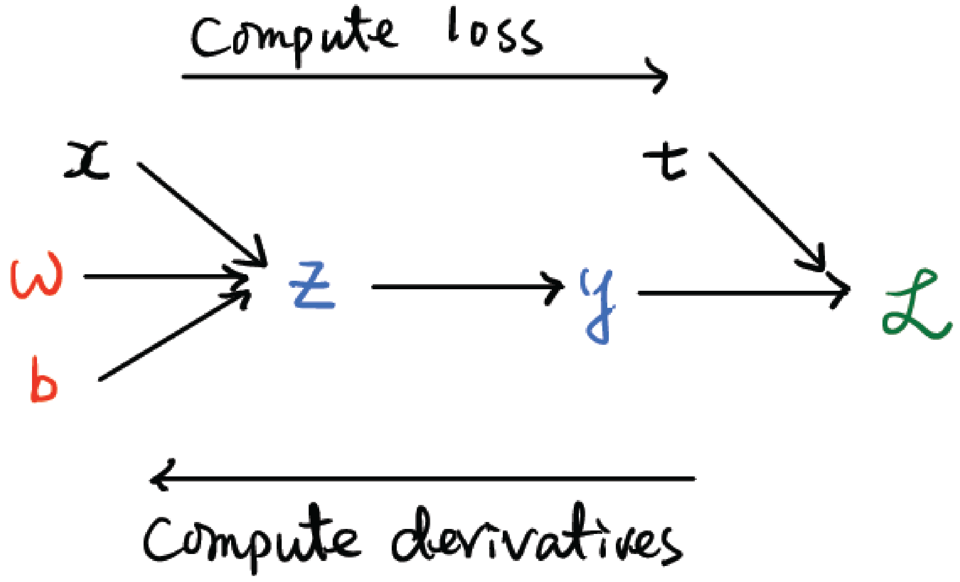
- Consider a univariate logistic least-square model $L = \frac{1}{2}(\varphi(wx + b) - t)^2$
- A more structural way

Compute the loss

$$z = wx + b$$
$$y = \varphi(z)$$
$$L = \frac{1}{2}(y - t)^2$$

Compute the derivatives:

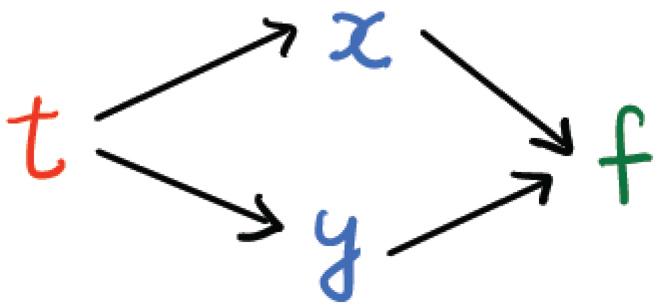
$$\bar{y} \equiv \frac{\partial L}{\partial y} = y - t$$
$$\bar{z} \equiv \frac{\partial L}{\partial z} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial z} = \bar{y}\,\varphi'(z)$$
$$\bar{w} \equiv \frac{\partial L}{\partial w} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w} = \bar{z}\,x$$
$$\bar{b} \equiv \frac{\partial L}{\partial b} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial b} = \bar{z}\;1$$



- $\bar{y}$, $\bar{z}$, $\bar{w}$ and $\bar{b}$ are computed by program

Back-Propagation Algorithm

- Multivariate chain rule



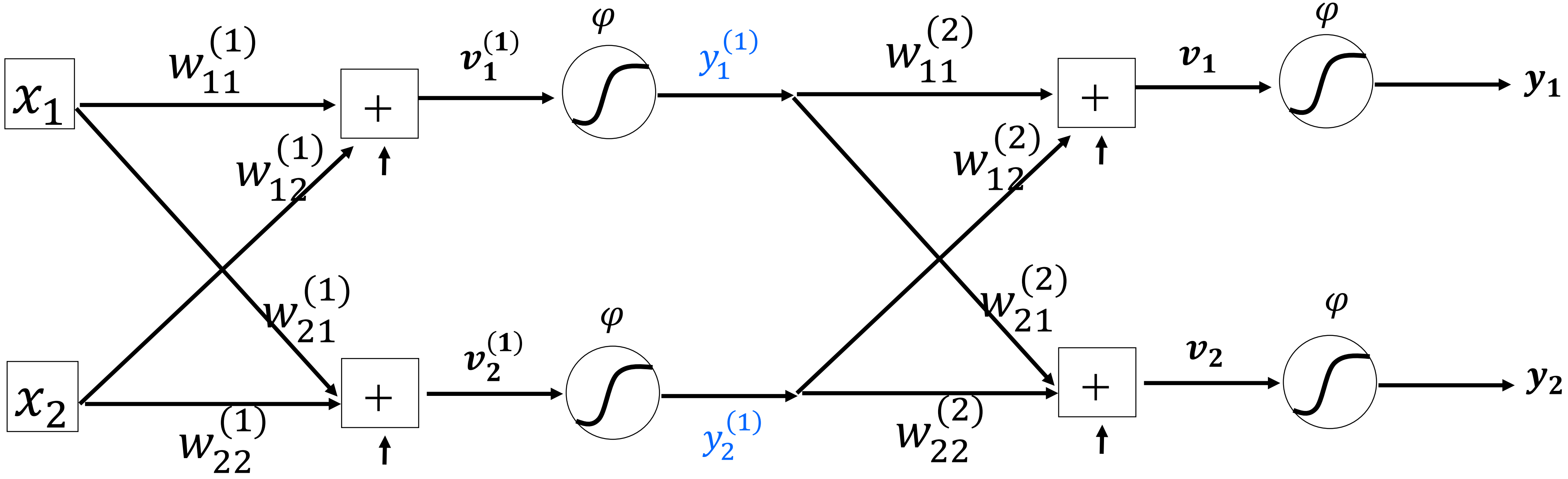
Mathematical expressions to be evaluated

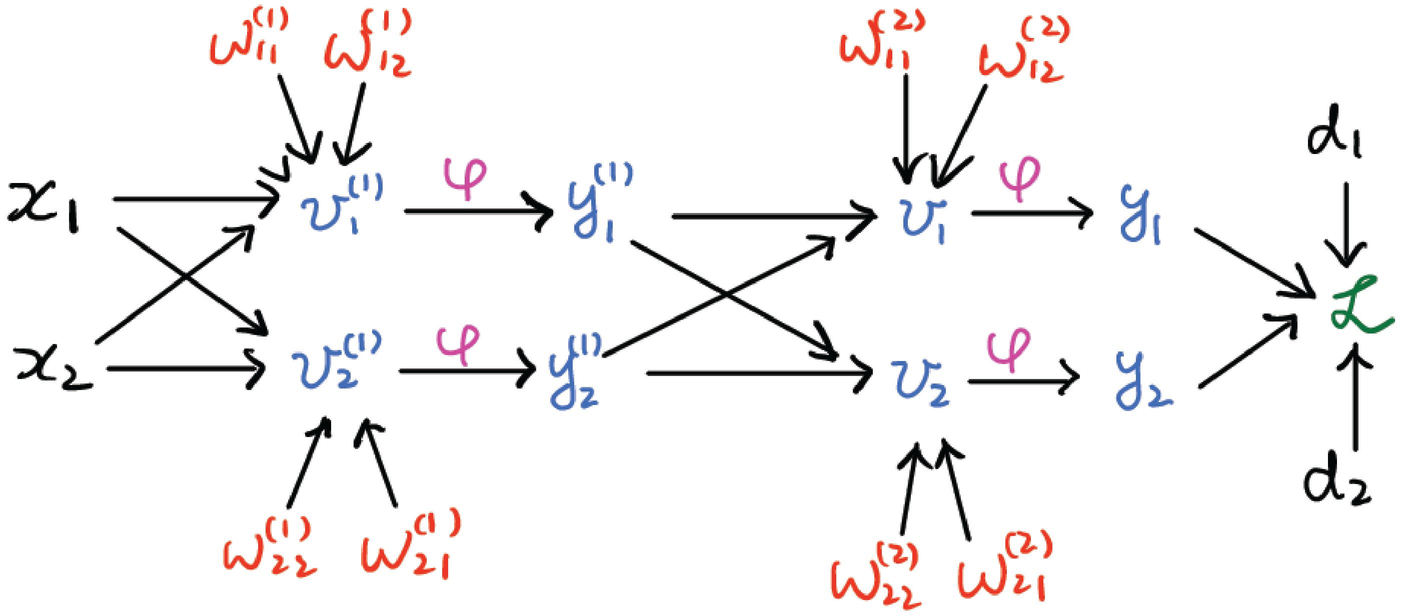
$$\frac{df}{dt} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

values already computed by program

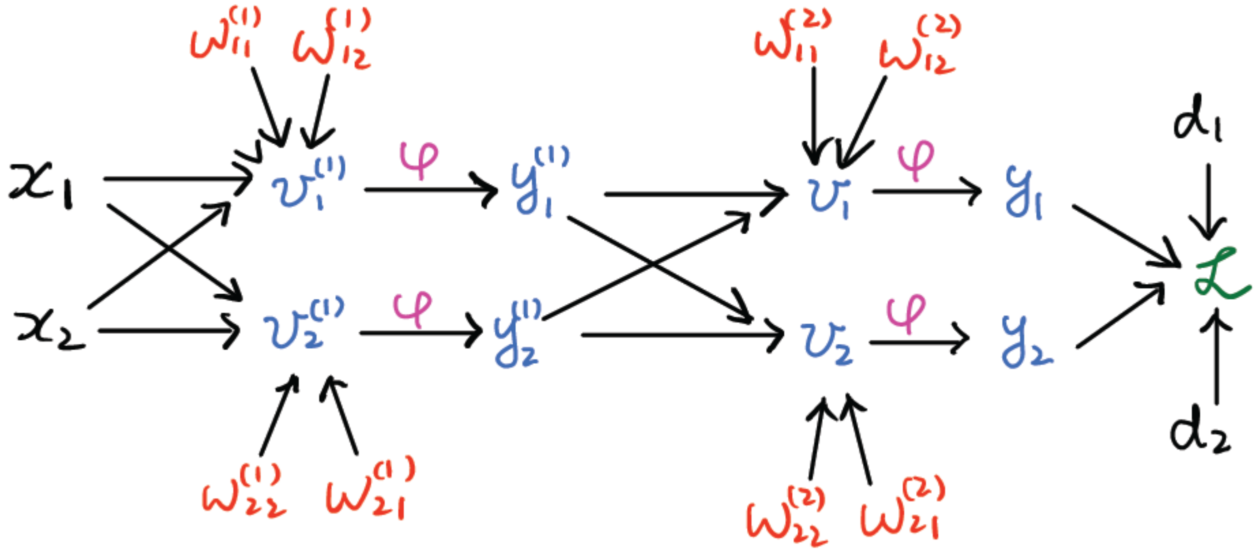
$$\bar{t} = \bar{x}\frac{dx}{dt} + \bar{y}\frac{dy}{dt}$$

Multi-layer Perceptron





Multi-layer Perceptron



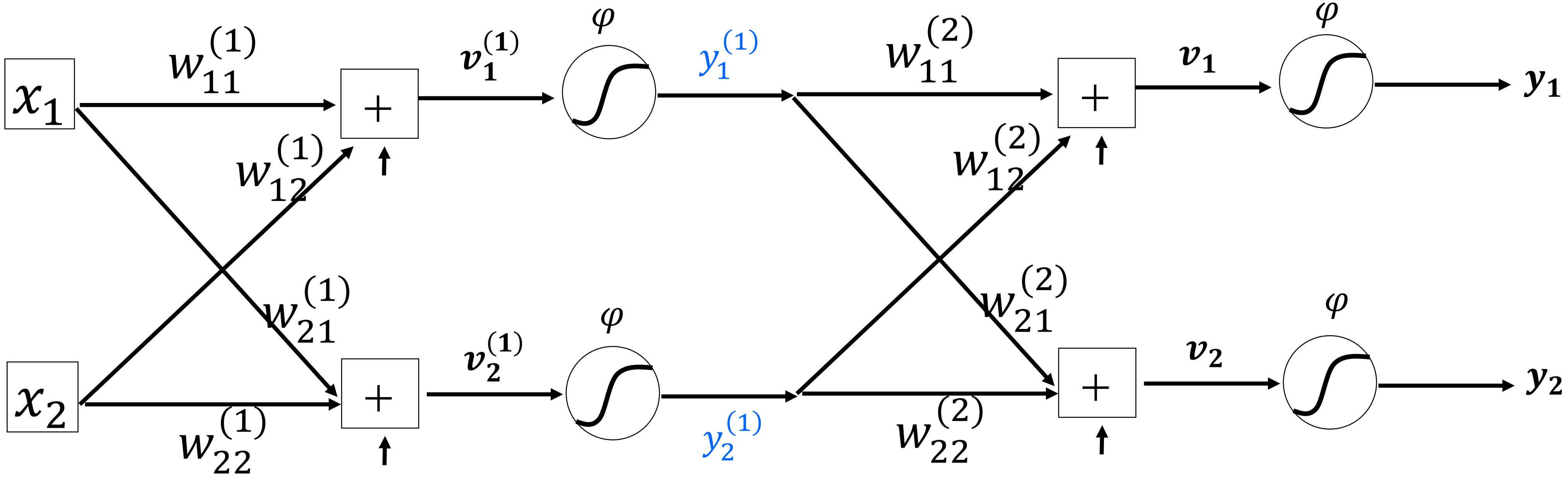
- Forward pass

$$v_j^{(1)} = \sum_{i=1}^{2} w_{ji}^{(1)} x_i$$
$$y_j^{(1)} = \varphi\left(v_j^{(1)}\right)$$
$$v_k = \sum_{j=1}^{2} w_{kj}^{(2)} y_j^{(1)}$$
$$y_k = \varphi(v_k)$$
$$L = \frac{1}{2}\sum_{k=1}^{2}(d_k - y_k)^2$$

- Backward pass

$$\overline{y_k} \equiv \frac{\partial L}{\partial y_k} = -(d_k - y_k) \equiv -e_k$$
$$\overline{v_k} \equiv \frac{\partial L}{\partial v_k} = \frac{\partial L}{\partial y_k}\frac{\partial y_k}{\partial v_k} = -e_k\varphi'(v_k) \equiv -\delta_k$$
$$\overline{w_{kj}^{(2)}} \equiv \frac{\partial L}{\partial w_{kj}^{(2)}} = \frac{\partial L}{\partial v_k}\frac{\partial v_k}{\partial w_{kj}^{(2)}} = -\delta_k y_j^{(1)}$$
$$\overline{y_j^{(1)}} \equiv \frac{\partial L}{\partial y_j^{(1)}} = \frac{\partial L}{\partial v_1}\frac{\partial v_1}{\partial y_j^{(1)}} + \frac{\partial L}{\partial v_2}\frac{\partial v_2}{\partial y_j^{(1)}}$$
$$= -\left(\delta_1 w_{1j}^{(2)} + \delta_2 w_{2j}^{(2)}\right) \equiv -e_j^{(1)}$$
$$\overline{v_j^{(1)}} \equiv \frac{\partial L}{\partial v_j^{(1)}} = \frac{\partial L}{\partial y_j^{(1)}}\frac{\partial y_j^{(1)}}{\partial v_j^{(1)}} = -e_j^{(1)}\varphi'\left(v_j^{(1)}\right) \equiv -\delta_j^{(1)}$$
$$\overline{w_{ji}^{(1)}} \equiv \frac{\partial L}{\partial w_{ji}^{(1)}} = \frac{\partial L}{\partial v_j^{(1)}}\frac{\partial v_j^{(1)}}{\partial w_{ji}^{(1)}} = -\delta_j^{(1)} x_j$$

Multi-layer Perceptron: forward pass



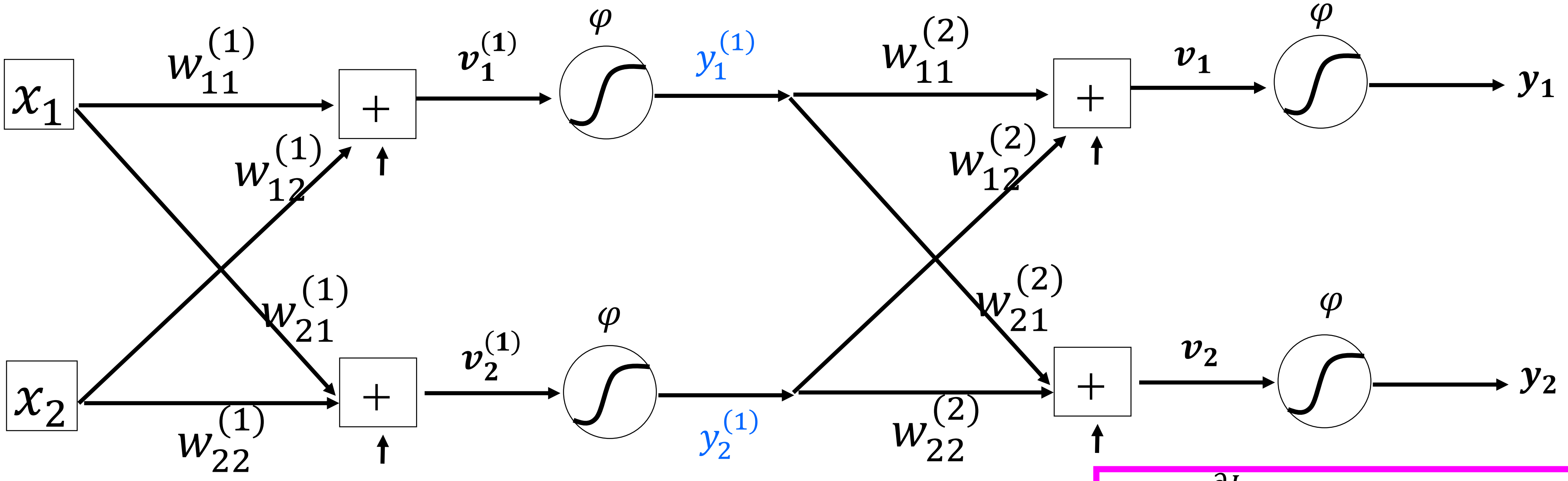
$$\begin{bmatrix} v_1^{(1)} \\ v_2^{(1)} \end{bmatrix} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \triangleq W_1 x \quad (Equation\ 3.1)$$

$$\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} w_{11}^{(2)} & w_{12}^{(2)} \\ w_{21}^{(2)} & w_{22}^{(2)} \end{bmatrix} \begin{bmatrix} y_1^{(1)} \\ y_2^{(1)} \end{bmatrix} \triangleq W_2 y^{(1)} \quad (Equation\ 3.2)$$

$$\begin{bmatrix} y_1^{(1)} \\ y_2^{(1)} \end{bmatrix} = \begin{bmatrix} \varphi\left(v_1^{(1)}\right) \\ \varphi\left(v_2^{(1)}\right) \end{bmatrix}$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \varphi(v_1) \\ \varphi(v_2) \end{bmatrix}$$

$$L = \frac{1}{2} \sum_{k=1}^{2} (d_k - y_k)^2$$



$$\overline{y_k} \equiv \frac{\partial L}{\partial y_k} = d_k - y_k \equiv -e_k$$

$$\overline{v_k} \equiv \frac{\partial L}{\partial v_k} = \frac{\partial L}{\partial y_k}\frac{\partial y_k}{\partial v_k} = -e_k \varphi'(v_k) \equiv -\delta_k$$

$$\overline{w_{kj}^{(2)}} \equiv \frac{\partial L}{\partial w_{kj}^{(2)}} = \frac{\partial L}{\partial v_k}\frac{\partial v_k}{\partial w_{kj}^{(2)}} = -\delta_k y_j^{(1)}$$

$$\overline{y_j^{(1)}} \equiv \frac{\partial L}{\partial y_j^{(1)}} = \frac{\partial L}{\partial v_1}\frac{\partial v_1}{\partial y_j^{(1)}} + \frac{\partial L}{\partial v_2}\frac{\partial v_2}{\partial y_j^{(1)}}$$

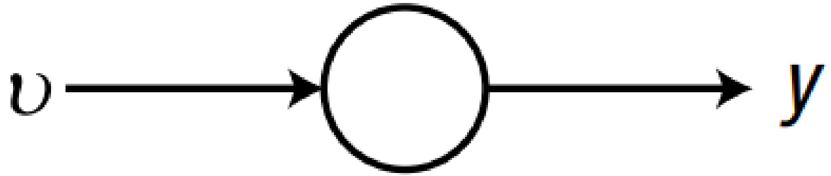
$$= -\left(\delta_1 w_{1j}^{(2)} + \delta_2 w_{2j}^{(2)}\right) \equiv -e_j^{(1)}$$

$$\overline{v_j^{(1)}} \equiv \frac{\partial L}{\partial v_j^{(1)}} = \frac{\partial L}{\partial y_j^{(1)}}\frac{\partial y_j^{(1)}}{\partial v_j^{(1)}} = -e_j^{(1)} \varphi'\left(v_j^{(1)}\right) \equiv -\delta_j^{(1)}$$

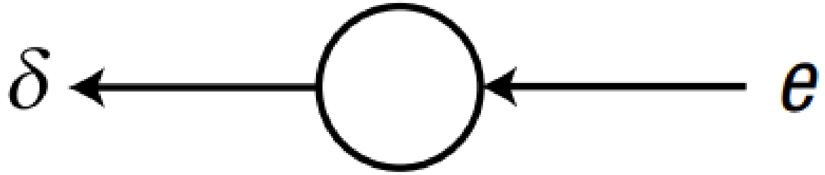
$$\overline{w_{ji}^{(1)}} \equiv \frac{\partial L}{\partial w_{ji}^{(1)}} = \frac{\partial L}{\partial v_j^{(1)}}\frac{\partial v_j^{(1)}}{\partial w_{ji}^{(1)}} = -\delta_j^{(1)} x_j$$

$$w_{kj}^{(2)} \leftarrow w_{kj}^{(2)} - \alpha \frac{\partial L}{\partial w_{kj}^{(2)}} = w_{kj}^{(2)} + \alpha \delta_k y_j^{(1)}$$

$$w_{ji}^{(1)} \leftarrow w_{ji}^{(1)} - \alpha \frac{\partial L}{\partial w_{ji}^{(1)}} = w_{ji}^{(1)} + \alpha \delta_j^{(1)} x_j$$



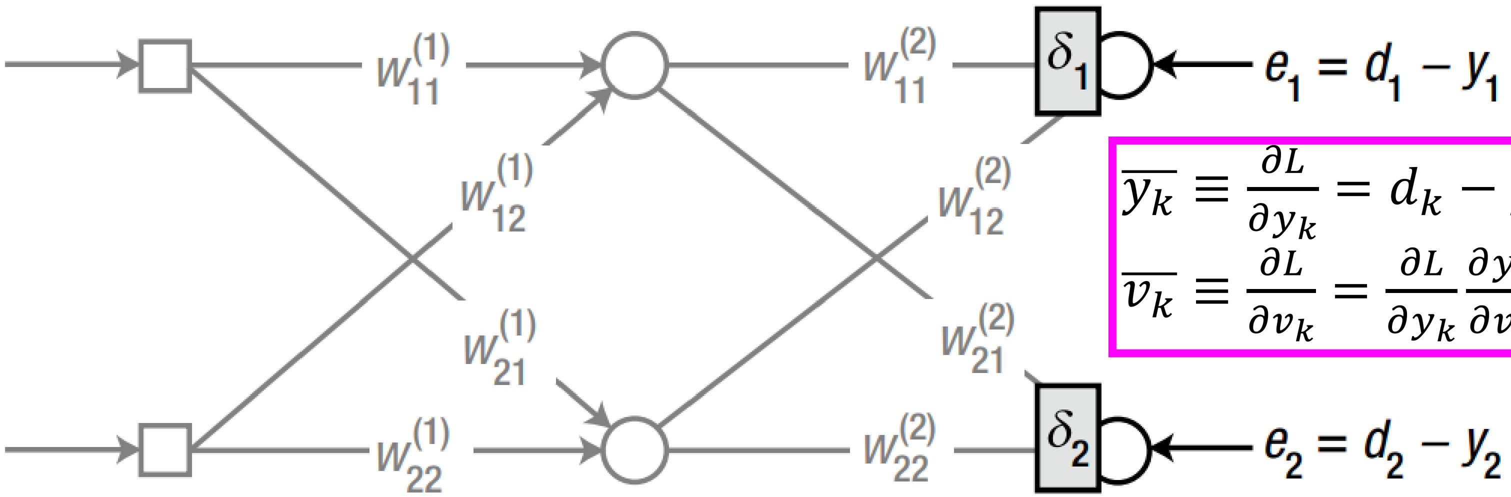
$$\varphi(\upsilon) = y$$



$$\delta = \varphi'(\upsilon) \times e$$

The forward and backward processes are identically applied to the hidden nodes as well as the output nodes.

The only difference is the error calculation.



$$\overline{y_k} \equiv \frac{\partial L}{\partial y_k} = d_k - y_k \equiv -e_k$$

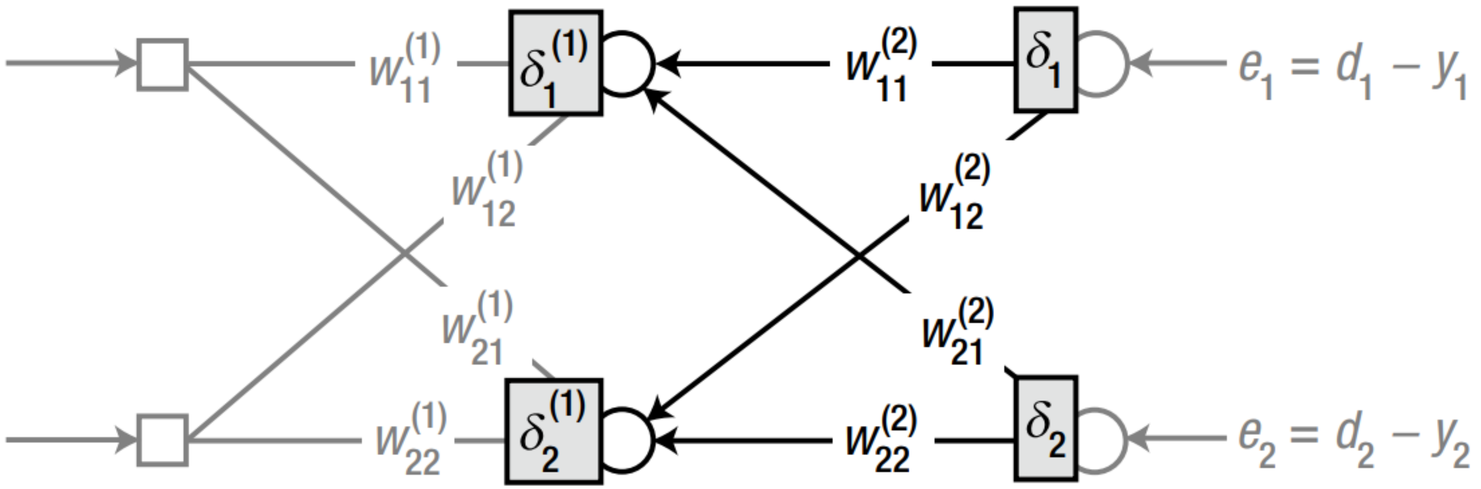
$$\overline{v_k} \equiv \frac{\partial L}{\partial v_k} = \frac{\partial L}{\partial y_k}\frac{\partial y_k}{\partial v_k} = -e_k\varphi'(v_k) \equiv -\delta_k$$

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

$$e_1 = d_1 - y_1 \qquad e_2 = d_2 - y_2$$

$$\delta_1 = \varphi'(v_1)e_1 \qquad \delta_2 = \varphi'(v_2)e_2 \qquad \text{(Equation 3.3)}$$

$\varphi'(\cdot)$ 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.



$$\overline{w_{kj}^{(2)}} \equiv \frac{\partial L}{\partial w_{kj}^{(2)}} = \frac{\partial L}{\partial v_k}\frac{\partial v_k}{\partial w_{kj}^{(2)}} = -\delta_k y_j^{(1)}$$

$$\overline{y_j^{(1)}} \equiv \frac{\partial L}{\partial y_j^{(1)}} = \frac{\partial L}{\partial v_1}\frac{\partial v_1}{\partial y_j^{(1)}} + \frac{\partial L}{\partial v_2}\frac{\partial v_2}{\partial y_j^{(1)}}$$

$$= -\left(\delta_1 w_{1j}^{(2)} + \delta_2 w_{2j}^{(2)}\right) \equiv -e_j^{(1)}$$

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

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

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

(Equation 3.4)

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

$$\Rightarrow \begin{bmatrix} e_1^{(1)} \\ e_2^{(1)} \end{bmatrix} = \begin{bmatrix} w_{11}^{(2)} & w_{21}^{(2)} \\ w_{12}^{(2)} & w_{22}^{(2)} \end{bmatrix} \begin{bmatrix} \delta_1 \\ \delta_2 \end{bmatrix} = W_2^T \begin{bmatrix} \delta_1 \\ \delta_2 \end{bmatrix}$$

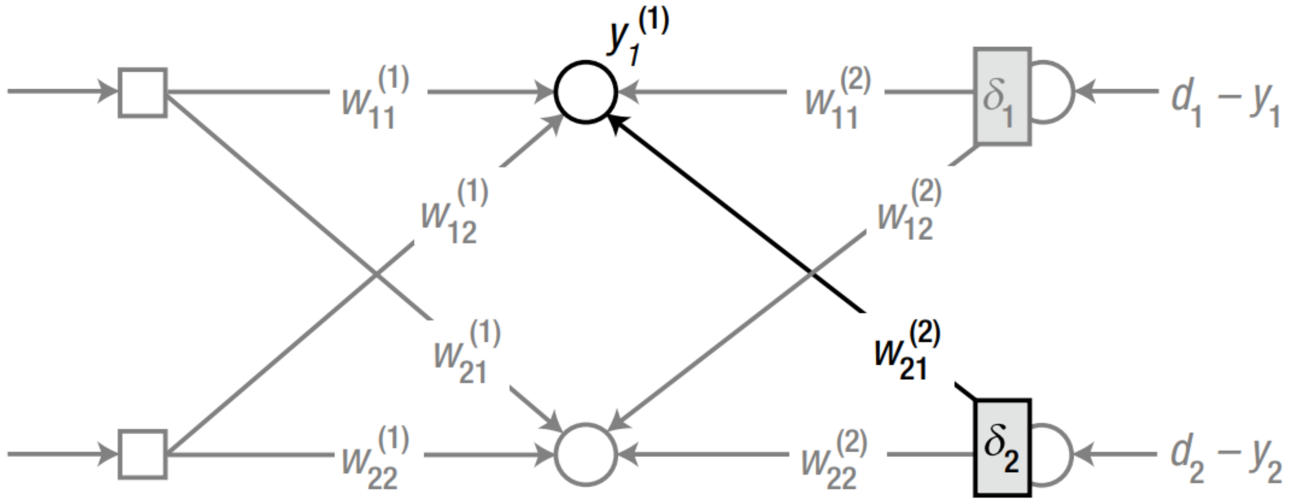
(Equation 3.5, 3.6)

- If we have additional hidden layers, we will just repeat the same backward process for each hidden layer and calculate all the deltas.

- Just use the following equation to adjust the weights of the respective layers.

$$\Delta w_{ij} = \alpha \delta_i x_j$$
$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \quad \text{(Equation 3.7)}$$

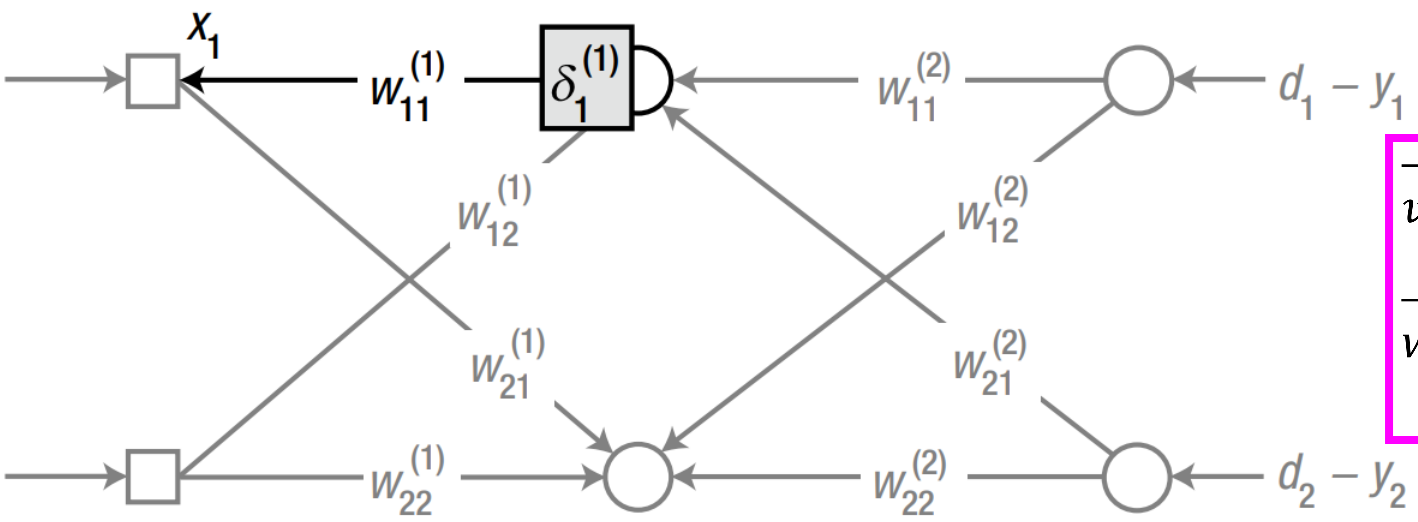
x_j is the input signal for the corresponding weight.
This equation is the same as that of the delta rule of the previous section.



- Consider the weight $w_{21}^{(2)}$ for example. The weight $w_{21}^{(2)}$ of figure can be adjusted using Equation 3.7 as:

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

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



$$\overline{v_j^{(1)}} \equiv \frac{\partial L}{\partial v_j^{(1)}} = \frac{\partial L}{\partial y_j^{(1)}}\frac{\partial y_j^{(1)}}{\partial v_j^{(1)}} = -e_j^{(1)}\varphi'\left(v_j^{(1)}\right) \equiv -\delta_j^{(1)}$$

$$\overline{w_{ji}^{(1)}} \equiv \frac{\partial L}{\partial w_{ji}^{(1)}} = \frac{\partial L}{\partial v_j^{(1)}}\frac{\partial v_j^{(1)}}{\partial w_{ji}^{(1)}} = -\delta_j^{(1)} x_j$$

- The weight $w_{11}^{(1)}$ of figure is adjusted using Equation 3.7 as：

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

x_1 is the output of the first input node.

Process to train the neural network using the backpropagation algorithm

1. Initialize the weights with adequate values.

2. Enter the input from the training data { input, correct output } and obtain the neural network's output.

3. Calculate the error of the output to the correct output and the delta, δ, of the output nodes.

$$e = d - y$$
$$\delta = \varphi'(v)e = \varphi(v)\big(1 - \varphi(v)\big)e$$

4. Propagate the output node delta, δ, backward, and calculate the deltas of the immediate next (left) nodes.

$$e^{(k)} = W^T \delta$$
$$\delta^{(k)} = \varphi'\left(v^{(k)}\right) e^{(k)}$$

5. Repeat Step 4 until it reaches the hidden layer that is on the immediate right of the input layer.

6. Adjust the weights according to the following learning rule.

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

7. Repeat Steps 2-5 for every training data point.

8. Repeat Steps 2-6 until the neural network is properly trained.

Example: Back-Propagation

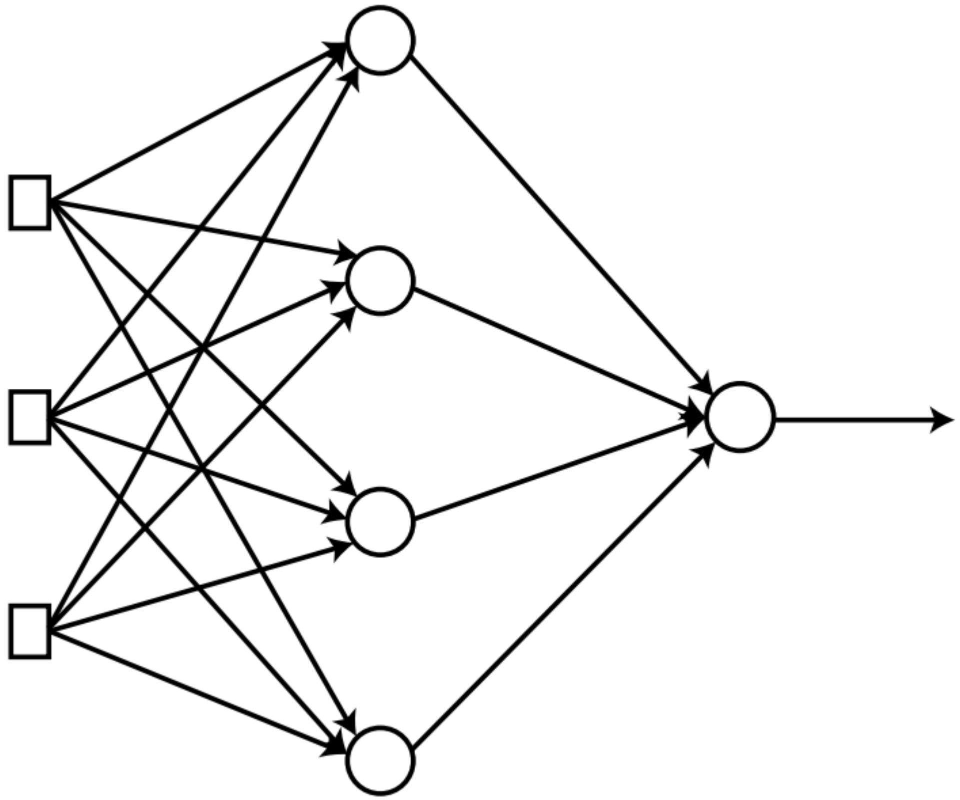
- The training data contains four elements. The red bolded rightmost number of the data is the correct output.

- This data is the one that the single-layer neural network had failed to learn.

- Ignoring the third value, the Z-axis, of the input, this dataset actually provides the **XOR logic operations**.

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

- Consider a neural network that consists of three input nodes and a single output node.
- It has one hidden layer of four nodes.
- The sigmoid function is used as the activation function for the hidden nodes and the output node.



XOR Problem

- The function BackpropXOR, which implements the back-propagation algorithm using the SGD method, takes the network's weights and training data and returns the adjusted weights.

$$[W_1, W_2] = BackpropXOR(W_1, W_2, X, D)$$

W_1 is the weight matrix between the input layer and hidden layer.
W_2 is the weight matrix between the hidden layer and output layer.
X and D are the input and correct output of the training data, respectively.

- The code takes point from the training dataset, calculates the weight update, dW, using the delta rule, and adjusts the weights.

- The delta (delta1) calculation using the back-propagation of the output delta as follows：

e1 = W2’ * delta;
delta1 = y1 .* (1 - y1) .* e1;

The calculation of the error, e1, is the implementation of Equation 3.6.

```
function [W1, W2] = BackpropXOR(W1, W2, X, D)
  alpha = 0.9;

  N = 4;
  for k = 1:N
    x = X(k, :)';
    d = D(k);

    v1 = W1*x;
    y1 = Sigmoid(v1);
    v  = W2*y1;
    y  = Sigmoid(v);

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

    e1     = W2'*delta;
    delta1 = y1.*(1-y1).*e1;

    dW1 = alpha*delta1*x';
    W1  = W1 + dW1;

    dW2 = alpha*delta*y1';
    W2  = W2 + dW2;
  end
end
```

- The function Sigmoid, which the BackpropXOR code calls, replaced the division with the element-wise division “ ./ ” to account for the vector.

```
function y = Sigmoid(x)
  y = 1 ./ (1 + exp(-x));
end
```

- The modified Sigmoid function can operate using vectors as shown by the following example：

$$Sigmoid(\ [-1, 0, 1]\ ) \ \rightarrow \ [0.2689, \quad 0.5000, \quad 0.7311]$$

- Execute the code, and find the following values are very close to the correct output, D, indicating that the neural network has been properly trained.

$$\begin{bmatrix} 0.0060 \\ 0.9888 \\ 0.9891 \\ 0.0134 \end{bmatrix} \Leftrightarrow \quad D = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

```
clear all

X = [ 0 0 1;
      0 1 1;
      1 0 1;
      1 1 1];

D = [ 0; 1; 1; 0 ];

W1 = 2*rand(4, 3) - 1;
W2 = 2*rand(1, 4) - 1;

for epoch = 1:10000           % train
  [W1, W2] = BackpropXOR(W1, W2, X, D);
end

N = 4;                        % inference
for k = 1:N
  x  = X(k, :)';
  v1 = W1*x;
  y1 = Sigmoid(v1);
  v  = W2*y1;
  y  = Sigmoid(v)
end
```

Momentum

- The momentum, *m*, is a term that is **added to the delta rule for adjusting the weight.**
- The use of the momentum term drives the weight adjustment to a certain direction to some extent, **rather than producing an immediate change**.
- It acts **similarly to physical momentum**, which impedes the reaction of the body to the external forces.

$$\Delta w_{ij} = \alpha \delta_i x_j$$
$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$$
(Equation 3.7)

$$\Delta \mathrm{w} = \alpha \delta x$$
$$m = \Delta w + \beta m^-$$
$$w \leftarrow w + m$$
$$m^- = m$$
(Equation 3.8)

- m^- is the previous momentum and β is a positive constant that is less than 1.
- The following steps show how the momentum changes over time：

$$
\begin{aligned}
m(0) &= 0 \\
m(1) &= \Delta w(1) + \beta m(0) = \Delta w(1) \\
m(2) &= \Delta w(2) + \beta m(1) = \Delta w(2) + \beta \Delta w(1) \\
m(3) &= \Delta w(3) + \beta m(2) = \Delta w(3) + \beta\{\Delta w(2) + \beta \Delta w(1)\} \\
&= \Delta w(3) + \beta \Delta w(2) + \beta^2 \Delta w(1) \\
&\vdots
\end{aligned}
$$

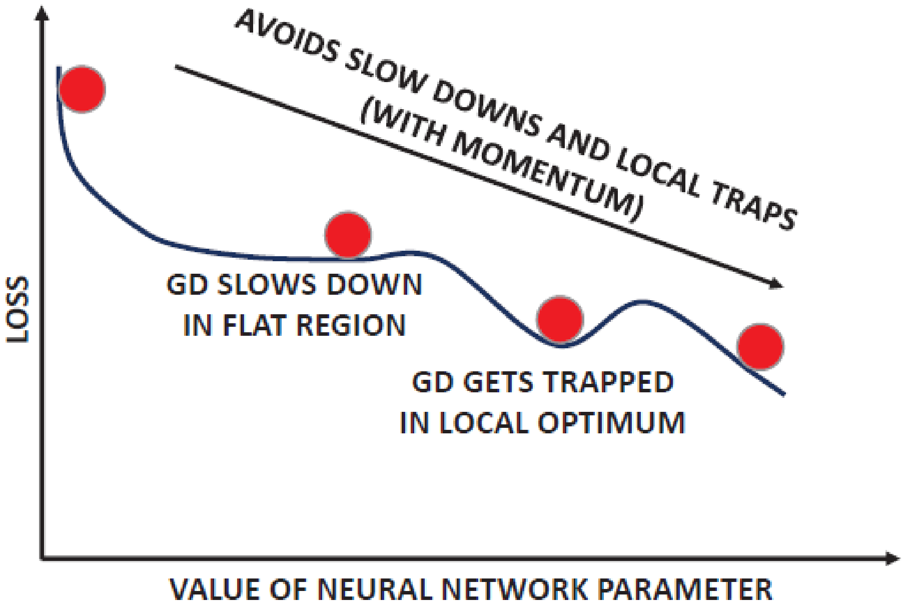
- It is noticeable from these steps that the previous weight update, i.e. $\Delta w(1)$, $\Delta w(2)$, $\Delta w(3)$, etc., is added to each momentum over the process.

- Since **β is less than 1**, the older weight update exerts a lesser influence on the momentum.

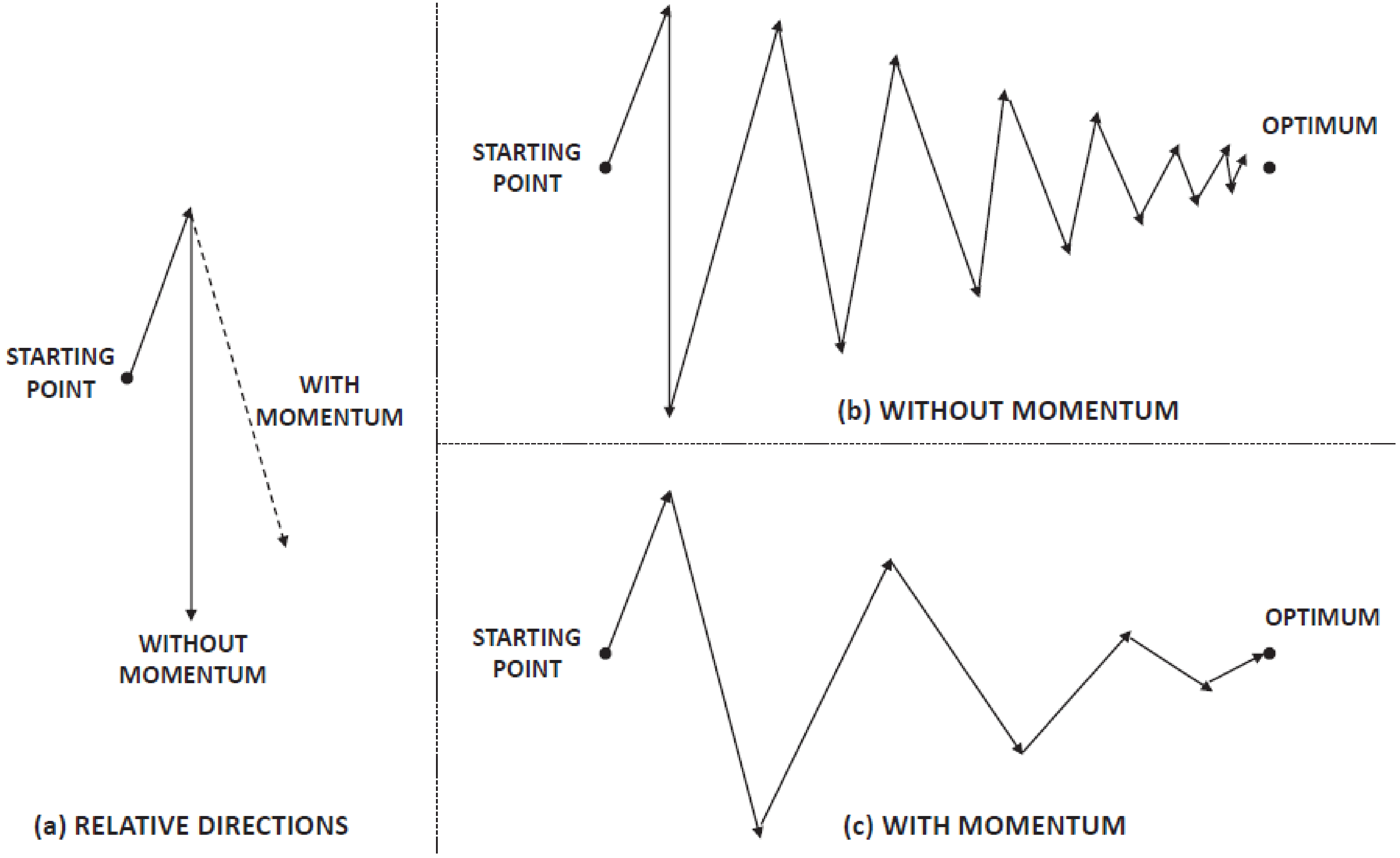
- Although the influence diminishes over time, **the old weight updates remain in the momentum**.
  - Therefore, the weight is not solely affected by a particular weight update value.
  - Therefore, **the learning stability improves**.

- In addition, the momentum grows more and more with weight updates. As a result, the weight update becomes greater and greater as well. Therefore, the learning rate increases.

Marble Rolling Down Hill



Avoiding Zig-Zagging with Momentum



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

- BackpropMmt.m file implements the back-propagation algorithm with the momentum.

- The code initializes the momentums, mmt1 and mmt2, as zeros when it starts the learning process.

- The weight adjustment formula is modified to reflect the momentum as：

dW1 = alpha * delta1 * x’;
Mmt1 = dW1 + beta * mmt1;
W1 = W1 + mmt1;

```
[W1 W2] = BackpropMmt(W1, W2, X, D)
function [W1, W2] = BackpropMmt(W1, W2, X, D)
  alpha = 0.9;
  beta  = 0.9;

  mmt1 = zeros(size(W1));
  mmt2 = zeros(size(W2));

  N = 4;
  for k = 1:N
    x = X(k, :)';
    d = D(k);

    v1 = W1*x;
    y1 = Sigmoid(v1);
    v  = W2*y1;
    y  = Sigmoid(v);

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

    e1     = W2'*delta;
    delta1 = y1.*(1-y1).*e1;

    dW1  = alpha*delta1*x';
    mmt1 = dW1 + beta*mmt1;
    W1   = W1 + mmt1;

    dW2  = alpha*delta*y1';
    mmt2 = dW2 + beta*mmt2;
    W2   = W2 + mmt2;
  end
end
```

- The TestBackpropMmt.m file tests the function BackpropMmt.

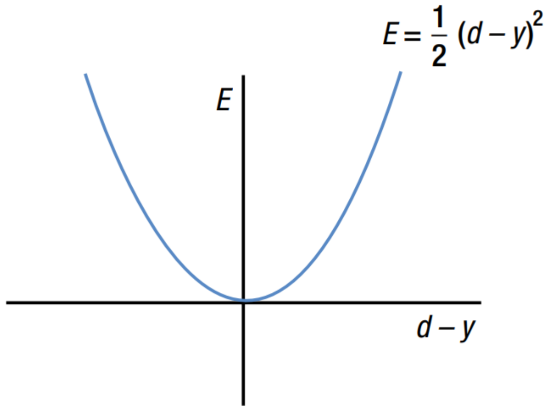
- The performance of the training is verified by comparing the output to the correct output of the training data.

```
clear all

X = [ 0 0 1;
      0 1 1;
      1 0 1;
      1 1 1 ];

D = [ 0; 1; 1; 0 ];

W1 = 2*rand(4, 3) - 1;
W2 = 2*rand(1, 4) - 1;

for epoch = 1:10000          % train
  [W1, W2] = BackpropMmt(W1, W2, X, D);
end

N = 4;                       % inference
for k = 1:N
  x  = X(k, :)';
  v1 = W1*x;
  y1 = Sigmoid(v1);
  v  = W2*y1;
  y  = Sigmoid(v)
end
```

Cost Function and Learning Rule

- There are two primary types of cost functions

$$L = \sum_{i=1}^{M} \frac{1}{2}(d_i - y_i)^2 \qquad (Equation\ 3.9)$$



$$L = \sum_{i=1}^{M} \{-d_i \ln(y_i) - (1-d_i)\ln(1-y_i)\} \qquad (Equation\ 3.10)$$

y_i is the output from the output node.
d_i is the correct output from the training data.
M is the number of output nodes.

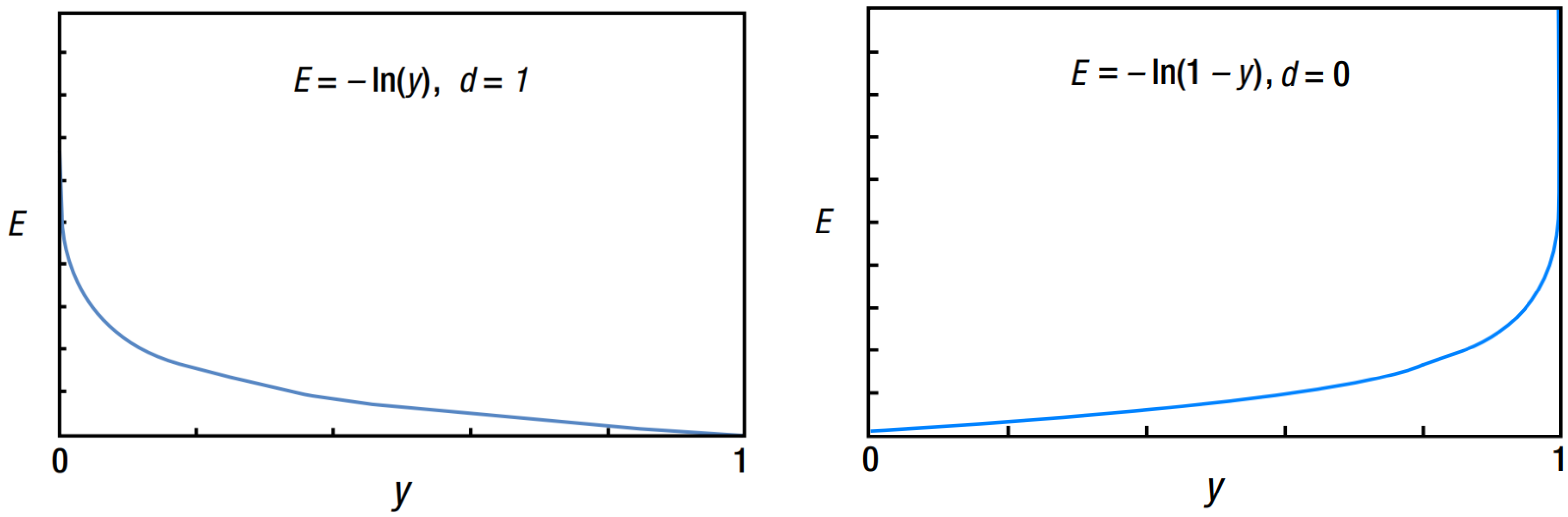
- Consider the following cost function which is called the **cross entropy** .

$$E = -d\ln(y) - (1-d)\ln(1-y) \qquad \text{Equation 3.10}$$

- Equation 3.10 is the concatenation of the following two equations：

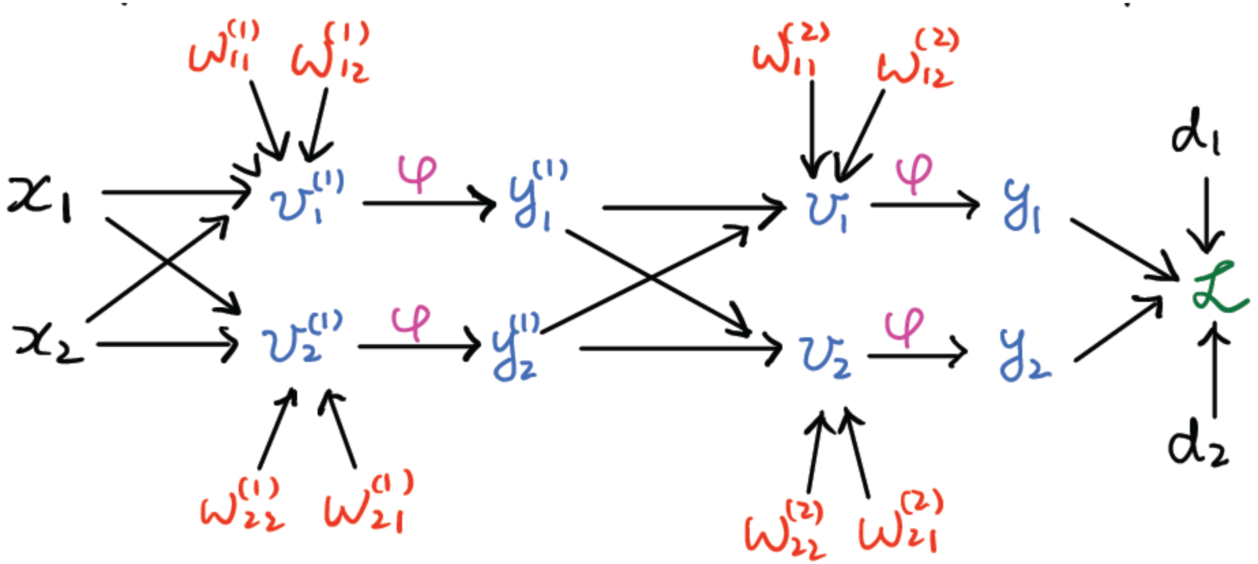
$$E = \begin{cases} -\ln(y) & d = 1 \\ -\ln(1-y) & d = 0 \end{cases}$$

- Due to the definition of a logarithm, **the output, y, should be within 0 and 1**. Therefore, the cross entropy cost function often teams up with **sigmoid** and **softmax** activation functions in the neural network.



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

Multi-layer Perceptron



- Backward pass

$$\overline{y_k} \equiv \frac{\partial L}{\partial y_k} = -(d_k - y_k)/y_k(1-y_k)$$

$$\overline{v_k} \equiv \frac{\partial L}{\partial v_k} = \frac{\partial L}{\partial y_k}\frac{\partial y_k}{\partial v_k} = -(d_k - y_k) \equiv -e_k \equiv -\delta_k$$

$$\overline{w_{kj}^{(2)}} \equiv \frac{\partial L}{\partial w_{kj}^{(2)}} = \frac{\partial L}{\partial v_k}\frac{\partial v_k}{\partial w_{kj}^{(2)}} = -\delta_k y_j^{(1)}$$

$$\overline{y_j^{(1)}} \equiv \frac{\partial L}{\partial y_j^{(1)}} = \frac{\partial L}{\partial v_1}\frac{\partial v_1}{\partial y_j^{(1)}} + \frac{\partial L}{\partial v_2}\frac{\partial v_2}{\partial y_j^{(1)}}$$

$$= -(\delta_1 w_{1j}^{(2)} + \delta_2 w_{2j}^{(2)}) \equiv -e_j^{(1)}$$

$$\overline{v_j^{(1)}} \equiv \frac{\partial L}{\partial v_j^{(1)}} = \frac{\partial L}{\partial y_j^{(1)}}\frac{\partial y_j^{(1)}}{\partial v_j^{(1)}} = -e_j^{(1)}\varphi'\left(v_j^{(1)}\right) \equiv -\delta_j^{(1)}$$

$$\overline{w_{ji}^{(1)}} \equiv \frac{\partial L}{\partial w_{ji}^{(1)}} = \frac{\partial L}{\partial v_j^{(1)}}\frac{\partial v_j^{(1)}}{\partial w_{ji}^{(1)}} = -\delta_j^{(1)} x_j$$

- Forward pass

$$v_j^{(1)} = \sum_{i=1}^{2} w_{ji}^{(1)} x_i$$

$$y_j^{(1)} = \varphi\left(v_j^{(1)}\right)$$

$$v_k = \sum_{j=1}^{2} w_{kj}^{(2)} y_j^{(1)}$$

$$y_k = \varphi(v_k)$$

$$L = \frac{1}{2}\sum_{k=1}^{2}(-d_k \ln(y_k) - (1-d_k)\ln(1-y_k))$$

$$\frac{\partial L}{\partial v_i} = \frac{\partial\left(\frac{1}{2}\sum_{k=1}^{2}(-d_k \ln(y_k) - (1-d_k)\ln(1-y_k))\right)}{\partial y_i}$$

$$= \frac{\partial(-d_i \ln(y_i) - (1-d_i)\ln(1-y_i))}{\partial y_i} = -d_i \frac{1}{y_i} - (1-d_i)\frac{1}{1-y_i} = \frac{-d_i(1-y_i) - (1-d_i)y_i}{y_i(1-y_i)}$$

$$= \frac{-(d_i - y_i)}{y_i(1-y_i)}$$

$$\frac{\partial L}{\partial v_i} = \frac{\partial\left(\frac{1}{2}\sum_{k=1}^{2}(-d_k \ln(y_k) - (1-d_k)\ln(1-y_k))\right)}{\partial v_i}$$

$$= \frac{\partial(-d_i \ln(y_i) - (1-d_i)\ln(1-y_i))}{\partial v_i}$$

$$= \frac{\partial(-d_i \ln(\varphi(v_i)) - (1-d_i)\ln(1-\varphi(v_i)))}{\partial v_i}$$

$$= -d_i \frac{1}{\varphi(v_i)} \frac{\partial \varphi(v_i)}{\partial v_i} - (1-d_i)\frac{1}{1-\varphi(v_i)}\frac{\partial \varphi(v_i)}{\partial v_i}$$

$$= -d_i \frac{1}{\varphi(v_i)} \varphi(v_i)\big(1-\varphi(v_i)\big) + (1-d_i)\frac{1}{1-\varphi(v_i)}\varphi(v_i)(1-\varphi(v_i))$$

$$= -d_i\big(1-\varphi(v_i)\big) + (1-d_i)\varphi(v_i)$$

$$= -d_i + \varphi(v_i)$$

$$= -(d_i - y_i)$$

The procedure in training the neural network with the sigmoid activation function at the output node using the **cross entropy**

(1)Initialize the neural network's weights with adequate values.

(2)Enter the input of the training data { input, correct output } to the neural network and obtain the output. Calculate the error, and calculate the delta, δ, of the output nodes.

$$e = d - y$$
$$\delta = e$$

(3) Propagate the delta of the output node backward and calculate the delta of the subsequent hidden nodes.

$$e^{(k)} = W^T \delta$$
$$\delta^{(k)} = \varphi'\left(v^{(k)}\right) e^{(k)}$$

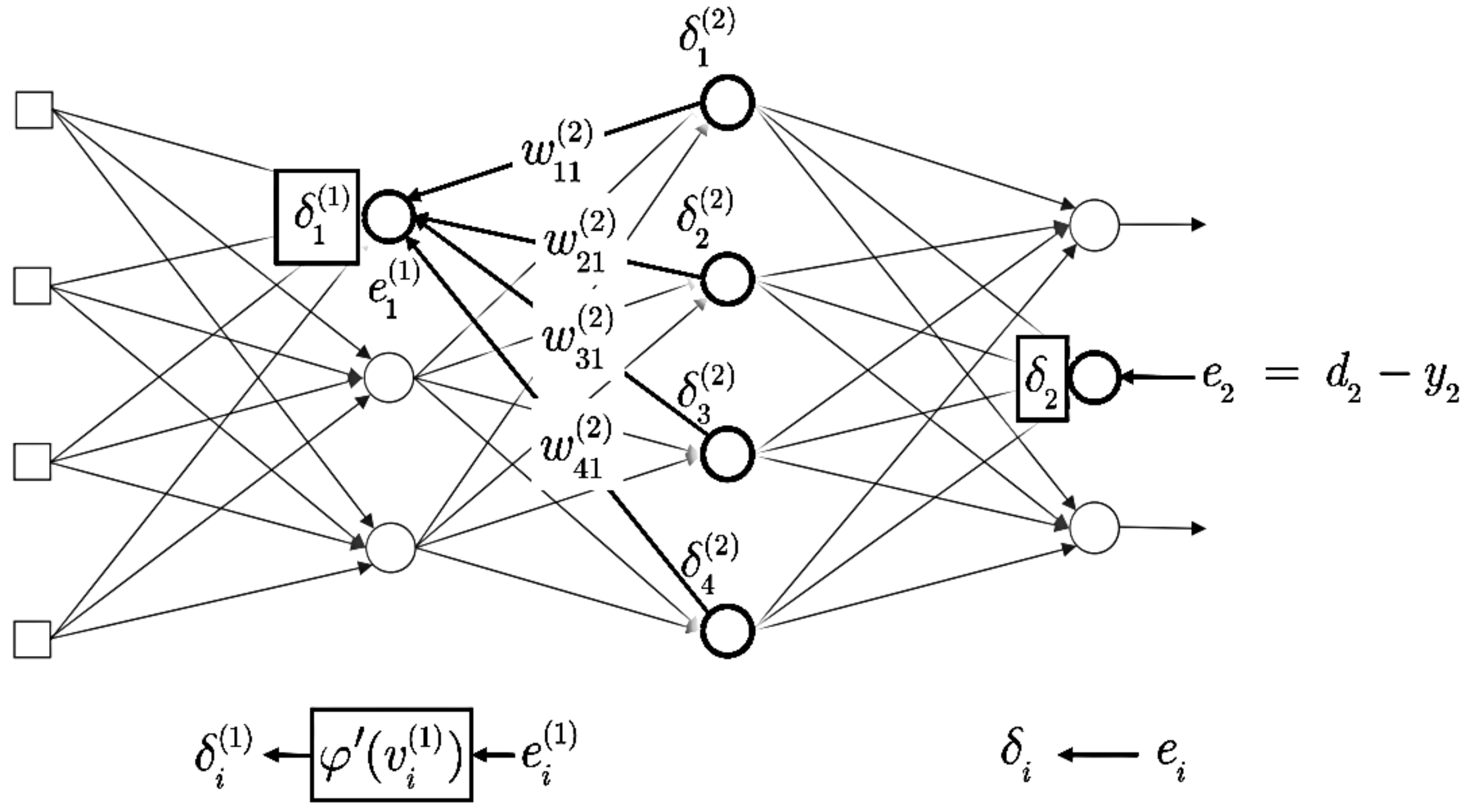
(4) Repeat Step 3 until it reaches the hidden layer that is next to the input layer.

(5) Adjust the neural network's weights using the following learning rule：

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

(6) Repeat Steps 2-5 for every training data point.

(7) Repeat Steps 2-6 until the network has been adequately trained.



- The output and hidden layers employ **different formulas of the delta** when the learning rule is based on the cross entropy and the sigmoid function.

Regularization

- Overfitting is a challenging problem that every technique of Machine Learning faces.

- 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**.

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

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

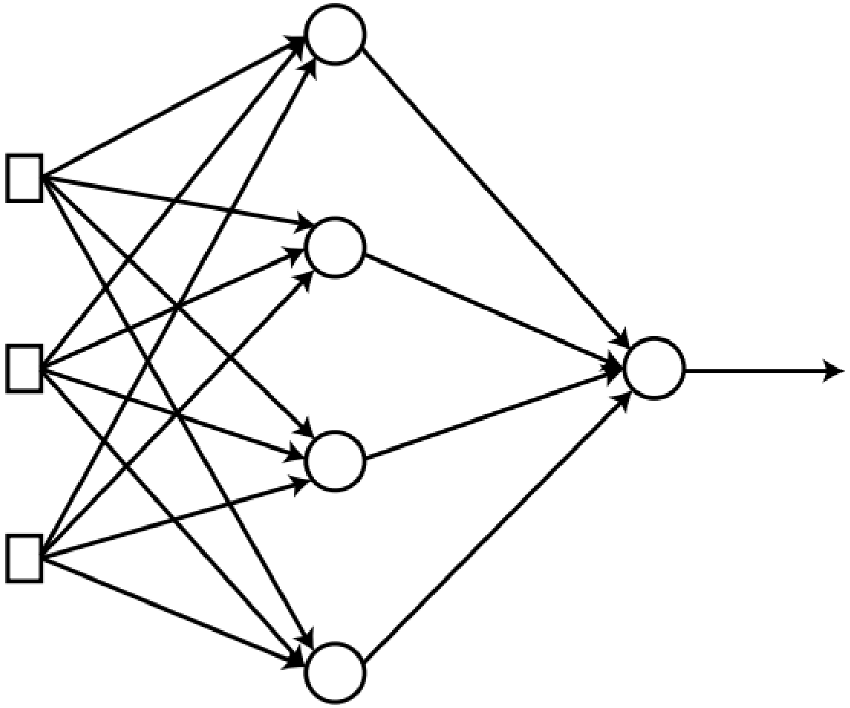
λ is the coefficient that determines how much of the connection weight is reflected on the cost function.

This cost function maintains a large value when one of the **output errors** and the **weight** remain large. Therefore, **solely making the output error zero will not suffice in reducing the cost function.**

In order to drop the value of the cost function, **both the error and weight should be controlled to be as small as possible.**

However, if a weight becomes small enough, the associated **nodes will be practically disconnected**. As a result, **unnecessary connections are eliminated, and the neural network becomes simpler.**

Example: Cross Entropy Function



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

The training data (XOR data) contains the same four elements.

- The sigmoid function is employed for the activation function of the hidden nodes and output node.

Cross Entropy Function

The BackpropCE function trains the XOR data using the cross entropy function.

It takes the neural network's weights and training data and returns the adjusted weights.

$$[W1, W2] = BackpropCE(W1, W2, X, D)$$

W1 and **W2** are the weight matrices for the input-hidden layers and hidden-output layers, respectively.

X and **D** are the input and correct output matrices of the data, respectively.

```
function [W1, W2] = BackpropCE(W1, W2, X, D)
  alpha = 0.9;

  N = 4;
  for k = 1:N
    x = X(k, :)';          % x = a column vector
    d = D(k);

    v1 = W1*x;
    y1 = Sigmoid(v1);
    v  = W2*y1;
    y  = Sigmoid(v);

    e     = d - y;
    delta = e;

    e1     = W2'*delta;
    delta1 = y1.*(1-y1).*e1;

    dW1 = alpha*delta1*x';
    W1 = W1 + dW1;

    dW2 = alpha*delta*y1';
    W2 = W2 + dW2;
  end
end
```

- This program calls the BackpropCE function and trains the neural network 10,000 times.

- The trained neural network yields the output for the training data input, and the result is

$$\begin{bmatrix} 0.00003 \\ 0.9999 \\ 0.9998 \\ 0.00036 \end{bmatrix} \Leftrightarrow \quad D = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

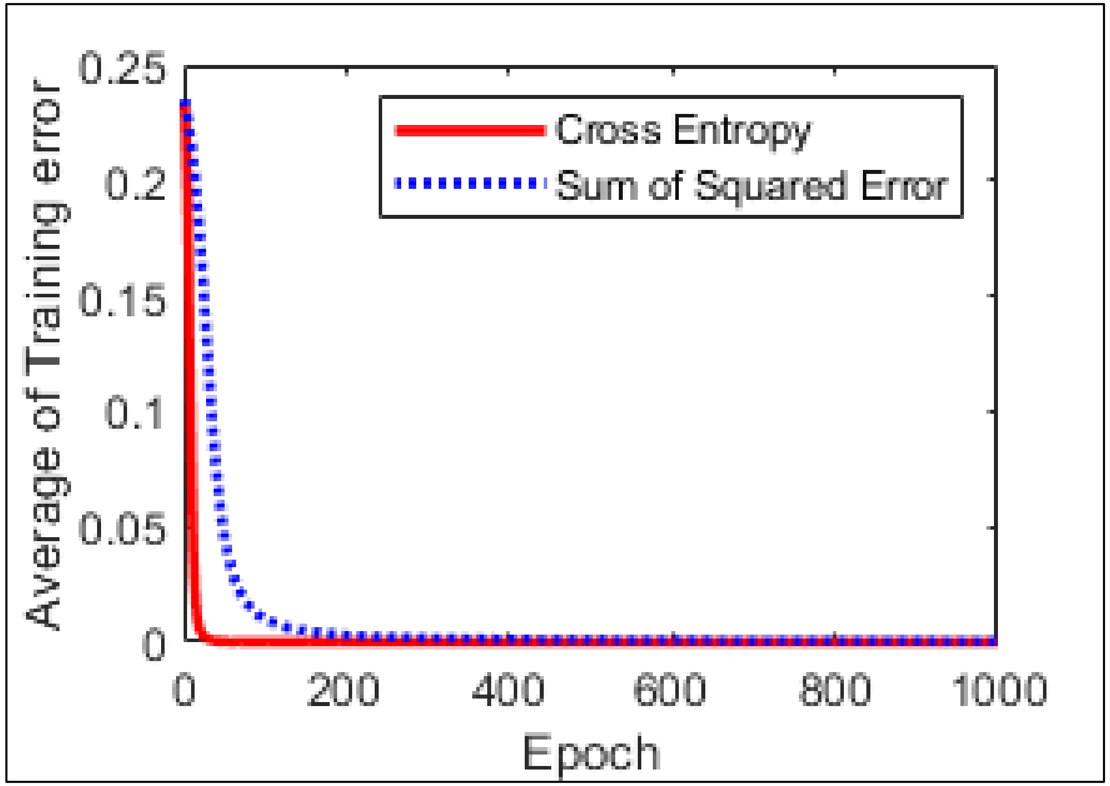
```
clear all

X = [ 0 0 1;
      0 1 1;
      1 0 1;
      1 1 1 ];

D = [ 0; 1; 1; 0 ];

W1 = 2*rand(4, 3) - 1;
W2 = 2*rand(1, 4) - 1;

for epoch = 1:10000           % train
  [W1, W2] = BackpropCE(W1, W2, X, D);
end

N = 4;                        % inference
for k = 1:N
  x  = X(k, :)';
  v1 = W1*x;
  y1 = Sigmoid(v1);
  v  = W2*y1;
  y  = Sigmoid(v)
end
```

Comparison of Cost Functions

The following listing shows the CEvsSSE.m file that compares the mean errors of the two functions.

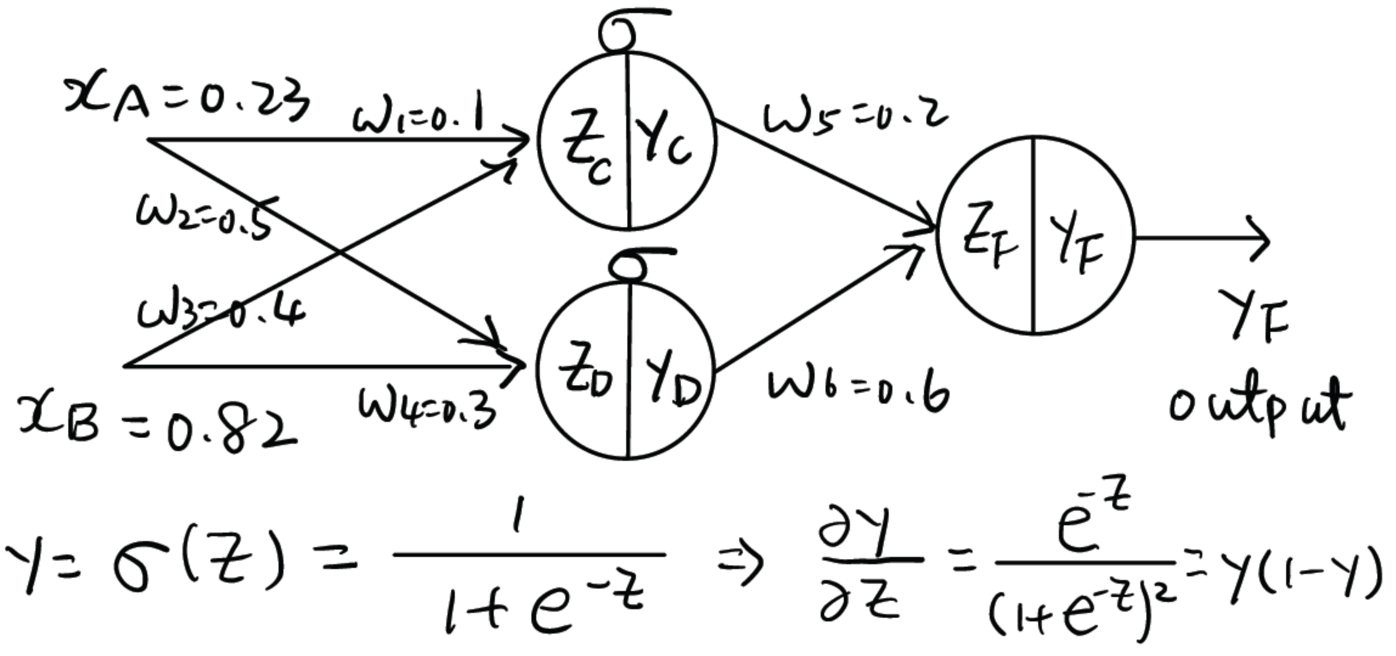
```
clear all

X = [ 0 0 1;
      0 1 1;
      1 0 1;
      1 1 1 ];

D = [ 0; 0; 1; 1 ];

E1 = zeros(1000, 1);
E2 = zeros(1000, 1);

W11 = 2*rand(4, 3) - 1;       % Cross entropy
W12 = 2*rand(1, 4) - 1;       %
W21 = W11;                    % Sum of squared error
W22 = W12;                    %
```

```
for epoch = 1:1000
  [W11, W12] = BackpropCE(W11, W12, X, D);
  [W21, W22] = BackpropXOR(W21, W22, X, D);

  es1 = 0;
  es2 = 0;
  N   = 4;
  for k = 1:N
    x = X(k, :)';
    d = D(k);

    v1  = W11*x;
    y1  = Sigmoid(v1);
    v   = W12*y1;
    y   = Sigmoid(v);
    es1 = es1 + (d - y)^2;

    v1  = W21*x;
    y1  = Sigmoid(v1);
    v   = W22*y1;
    y   = Sigmoid(v);
    es2 = es2 + (d - y)^2;
  end
  E1(epoch) = es1 / N;
  E2(epoch) = es2 / N;
end
```



- This program calls the BackpropCE and the BackpropXOR functions and trains the neural networks 1,000 times each.
- The squared sum of the output error (es1 and es2) is calculated at every epoch for each neural network, and their average (E1 and E2) is calculated.
- The cross entropy-driven training reduces the training error at a much faster rate.

Homework



1. $y_c = ?$ $y_D = ?$ $y_F = ?$
2. Compute mean square error $E = \frac{1}{2}(1 - y_F)^2$, 1 is the desired output
3. Using the training sample $x_A = 0.23, x_B = 0.82$ and the backward propagation algorithm for one iteration to compute $w_i \leftarrow w_i + \alpha \frac{\partial E}{\partial w_i}$, $\alpha = 0.7, i = 1,2,3,4,5,6$
4. Compute the forward pass using $x_A = 0.23, x_B = 0.82$ again and show that the mean square error has been reduced.