

---

# Data Mining & Machine Learning

Yong Zheng

Illinois Institute of Technology  
Chicago, IL, 60616, USA

**ILLINOIS TECH**

College of Computing

---

# Schedule

---

- Intro: Neural Networks
- Perceptron and Training
- Multi-layer Feed-forward Networks
- Backpropagation Training
- Neural Networks and Deep Learning

# Schedule

---

- Intro: Neural Networks
- Perceptron and Training
- Multi-layer Feed-forward Networks
- Backpropagation Training
- Neural Networks and Deep Learning

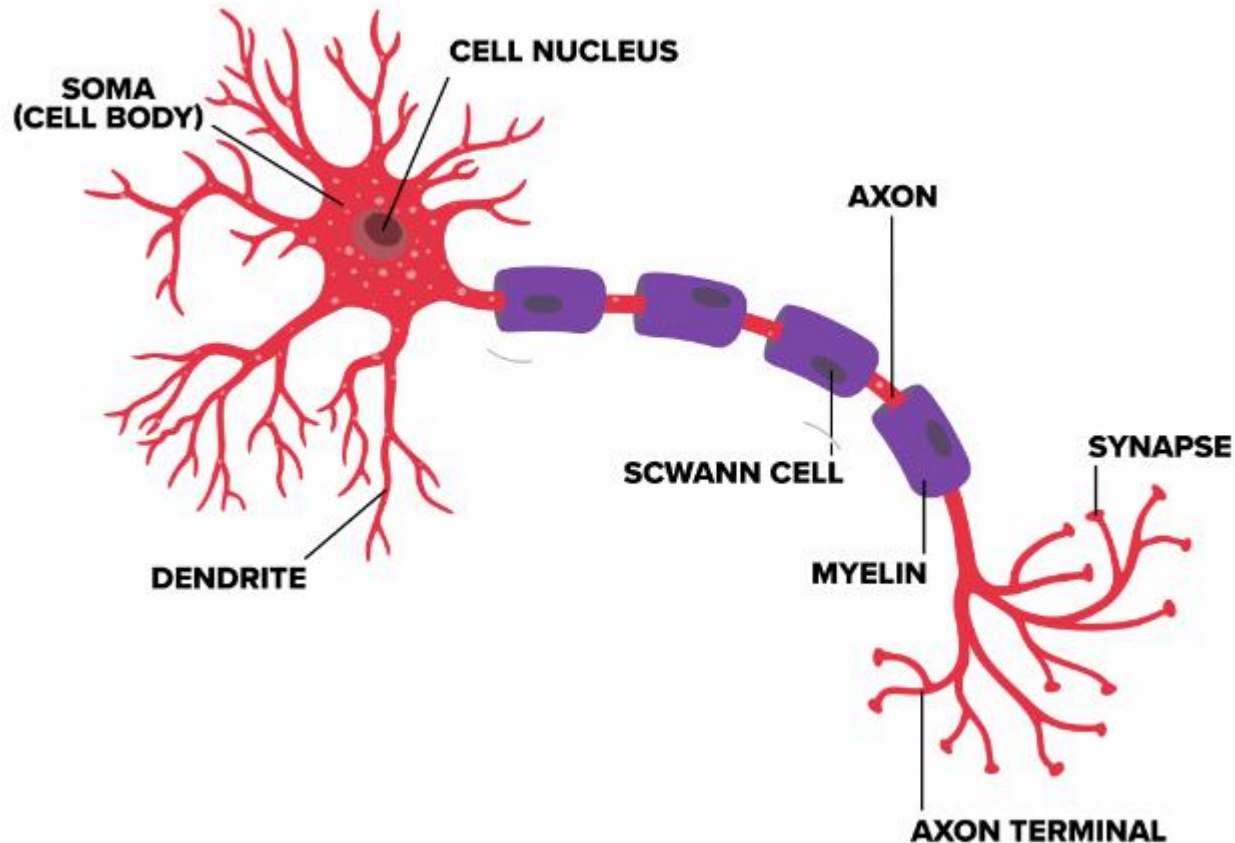
# Biological Neural Systems

---

- The original of neural networks is biological neural systems
  - Neuron switching time :  $> 10^{-3}$  secs
  - Number of neurons in the human brain:  $\sim 10^{10}$
  - Connections (synapses) per neuron :  $\sim 10^4 - 10^5$
  - Face recognition : 0.1 secs
  - High degree of distributed and parallel computation

# Biological Neural Systems

## NEURON ANATOMY



# Biological Neural Systems

---

- Different neurons are connected
- If one neuron is activated or exciting, it will send chemicals or signals to the connected neurons.
- If the degree of the excitement in one neuron is larger than a threshold, it will be activated and send chemicals or signals to other neurons

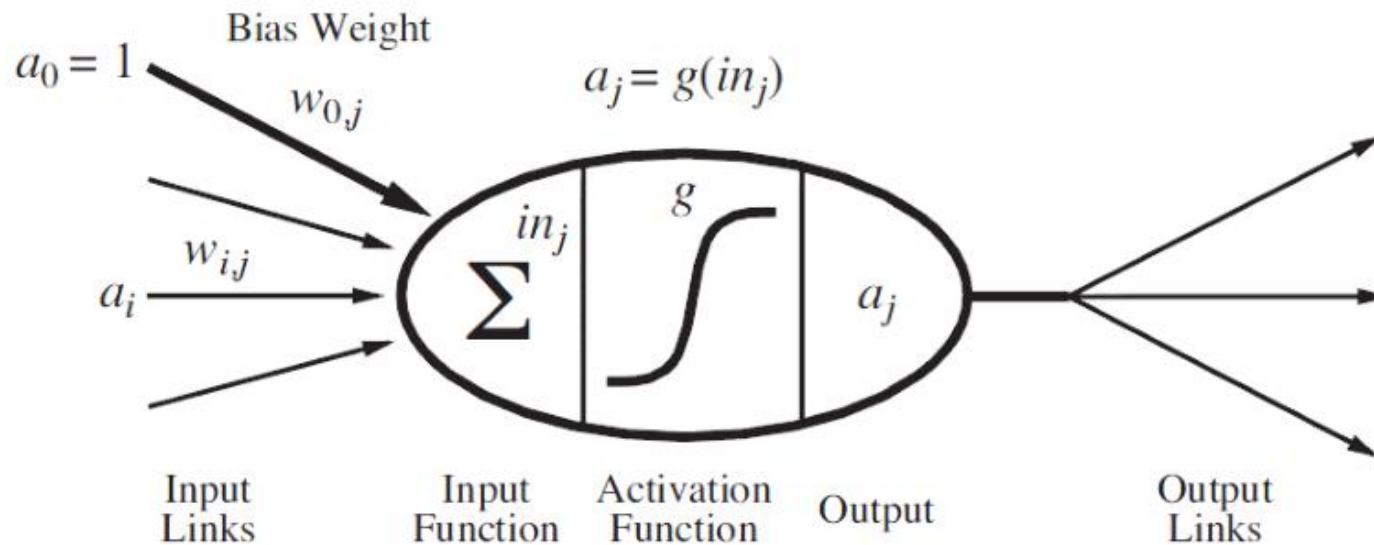
# Artificial Neural Networks (ANN)

---

- Researchers tried to learn from the biological neuron systems and built the ANN
  - There are many neuron units in ANN
  - They are connected within a structure
  - They work as threshold-switching units
  - There are weighted interconnections among units
  - We are able to learn and tune up these weights automatically by a training process

# Artificial Neural Networks (ANN)

- A neuron in ANN looks like this...



input signals  $\rightarrow$  input function(linear)  $\rightarrow$  activation function(nonlinear)  $\rightarrow$  output signal



# Schedule

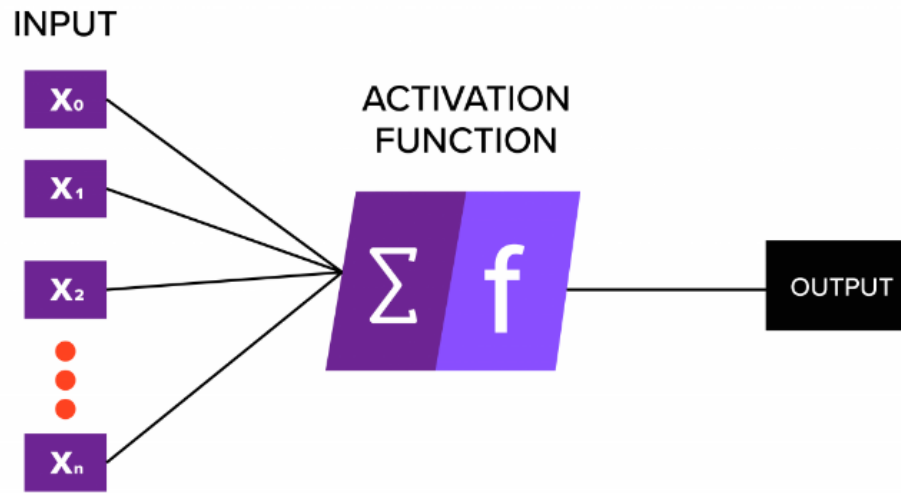
---

- Intro: Neural Networks
- Perceptron and Training
- Multi-layer Feed-forward Networks
- Backpropagation Training
- Neural Networks and Deep Learning

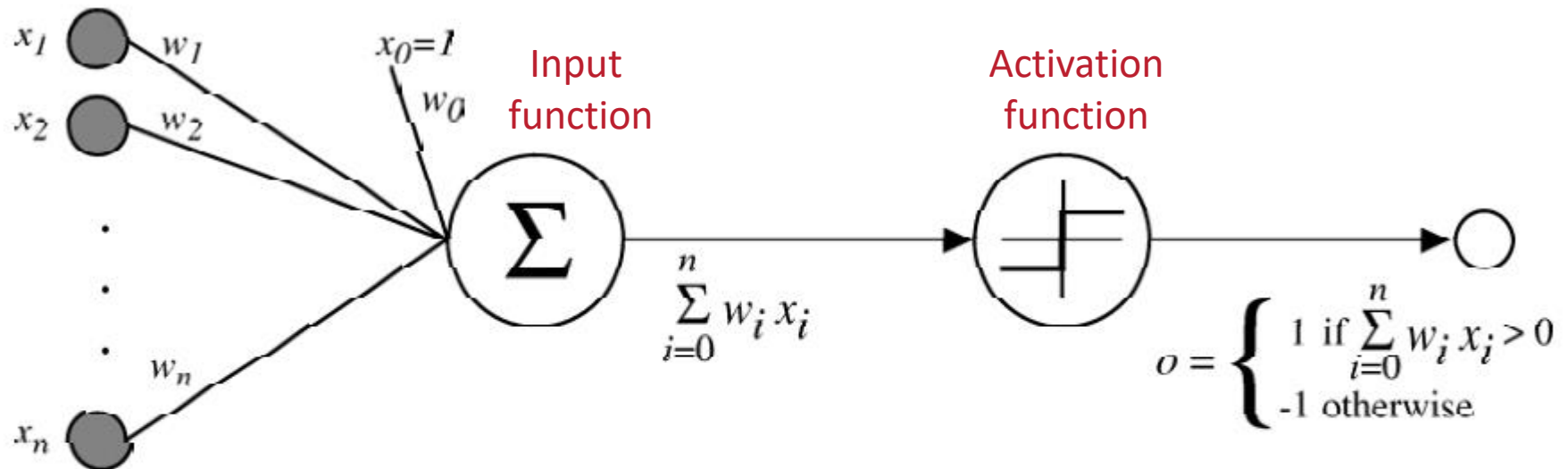
# Perceptron

- First neural network learning model in the 1960's
- Simple and limited (single layer models)
- Still used in current applications (modems, etc.)

## PERCEPTRON

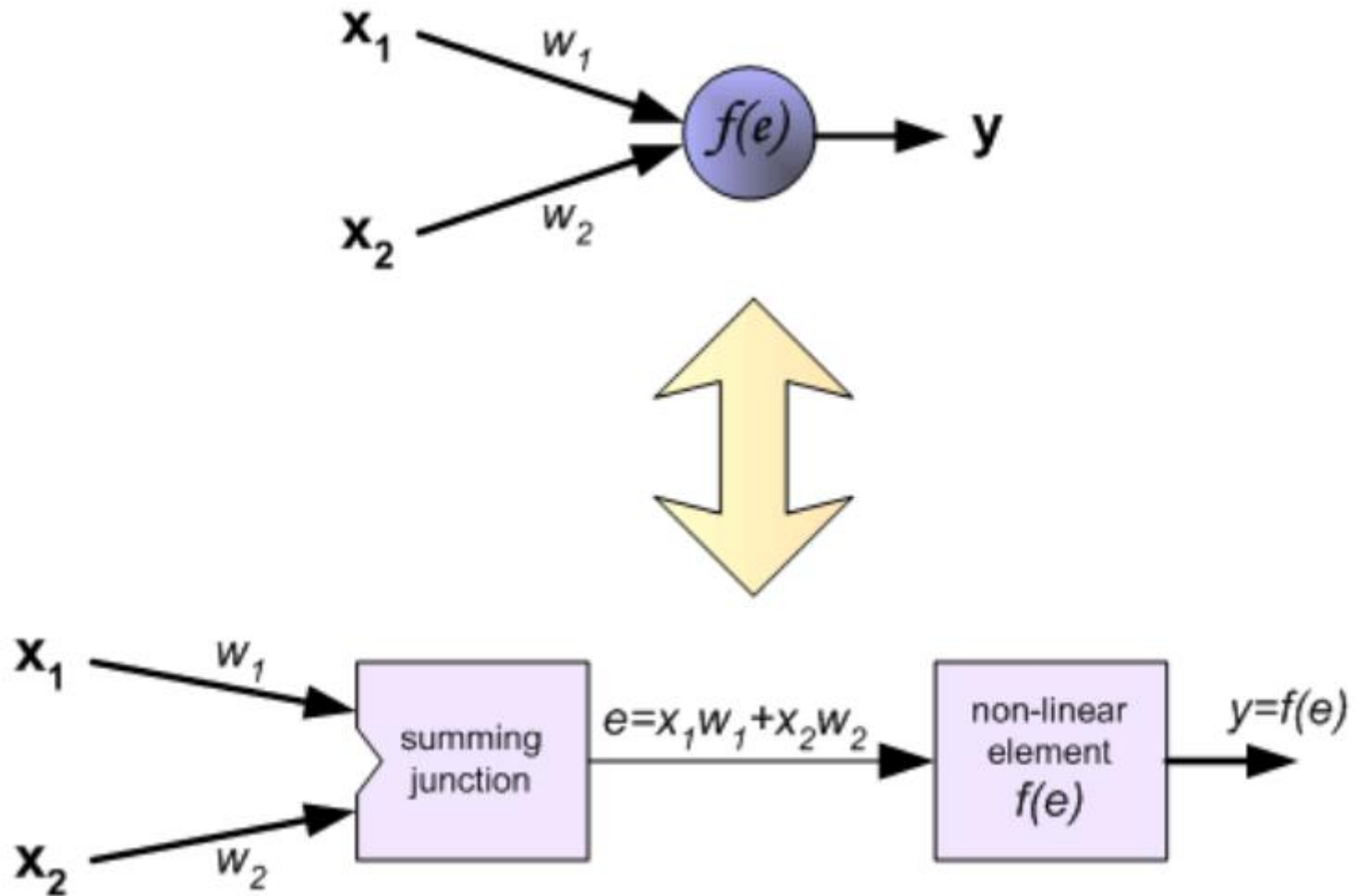


# Perceptron



$$o(x_1, \dots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \dots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

# Perceptron



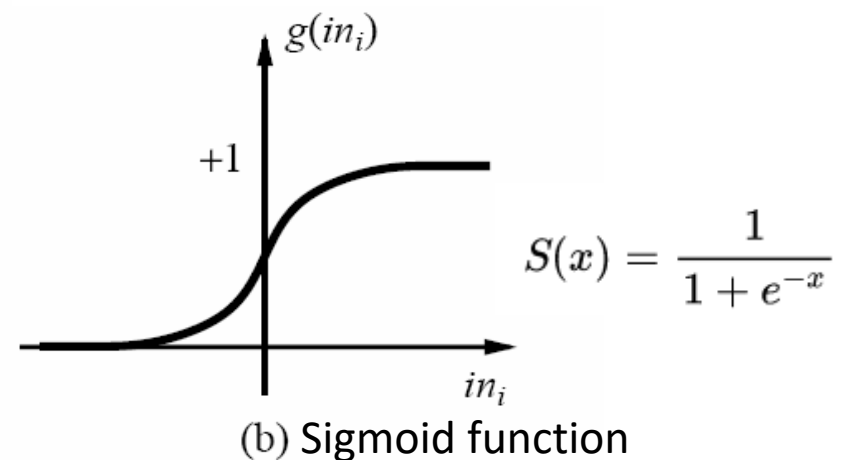
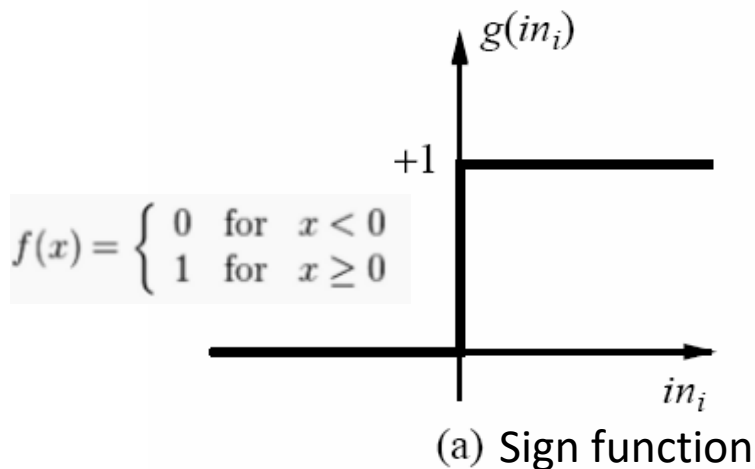
# Perceptron

---

- Input: different  $x$  variables with weights on edges
- Input function: it is used to aggregate the inputs, usually it is a weighted sum of its inputs
- Activation function
  - It is a threshold function
  - For the purpose of binary classification
    - The output is only 1 or 0
    - Sign function can be used as the activation function
    - Sigmoid can be used as activation function

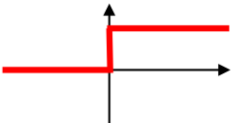
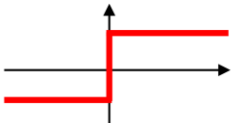
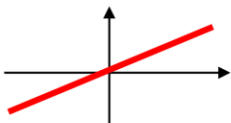
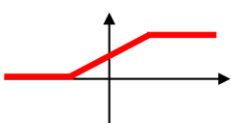
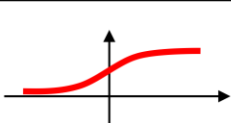
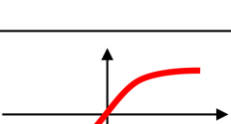
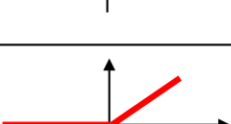

# Perceptron

- Activation function
  - Sign function
  - Sigmoid function
    - It is popular for classification, due to being easy to be updated and learned in the training process.

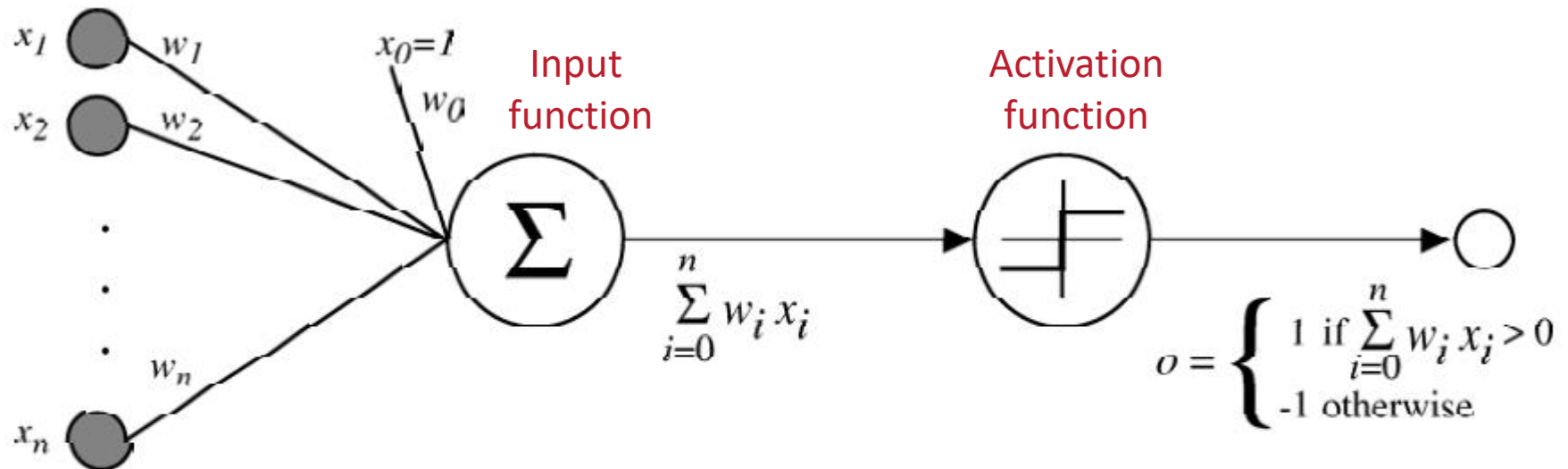


Neural networks can be used for both classifications and regressions.

It can be controlled by applying different activation functions.

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = \max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^z)$	Multi-layer Neural Networks	

# Perceptron

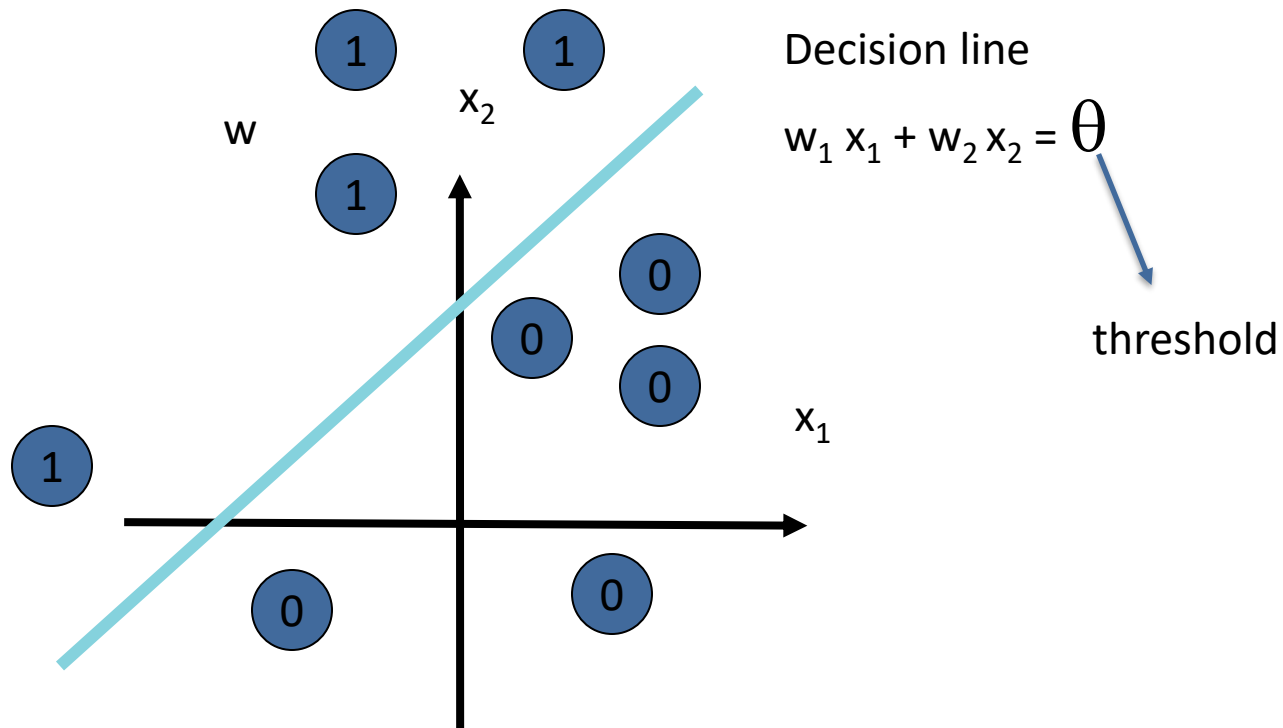


$$o(x_1, \dots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \dots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$



# Perceptron

- For classifications



# Perceptron

- Perceptron Training

- It is the simplest ANN model
- We need to train the model to learn the weights,  $w$

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

Update Rules

- $t$  is the real value
- $o$  is the output value (prediction by the model)
- $\eta$  is a constant value in  $[0, 1]$  as the learning rate

# Perceptron

- Perceptron Training
  - It is a process of iterative learning
    - At the beginning, give random values to  $w$
    - Get the output  $o$  through the perceptron
    - Update the  $w$  by using the update rules
    - Stop the learning process by a stopping criterion
      - Classification error is smaller than a threshold
      - Or, maximal learning iterations have been reached

We will use iterative learning in other techniques,  
similar to iterative learning in K-Means

# Perceptron: Example

- Consider learning the logical OR function

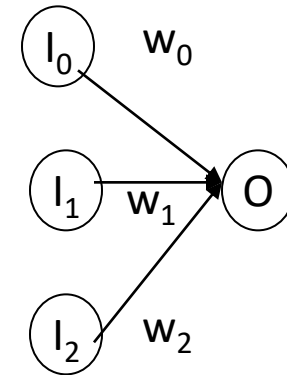
Sample	x0	x1	x2	label
1	1	0	0	0
2	1	0	1	1
3	1	1	0	1
4	1	1	1	1

- Activation function

$$S = \sum_{k=0}^{k=n} w_k x_k \quad S > 0 \text{ then } O = 1 \quad \text{else} \quad O = 0$$

# Perceptron: Example

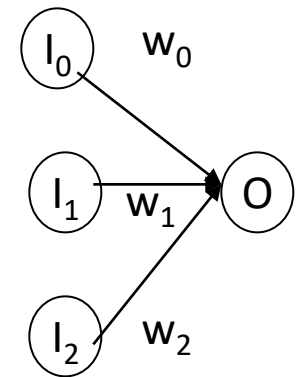
- We'll use a single perceptron with three inputs.
- We'll start with all weights 0  $W = \langle 0, 0, 0 \rangle$
- Example 1  $I = \langle 1 \ 0 \ 0 \rangle$  label=0  $W = \langle 0, 0, 0 \rangle$
- Perceptron ( $1 \times 0 + 0 \times 0 + 0 \times 0 = 0$ ,  $S=0$ ) output  $\rightarrow 0$   
 $\rightarrow$  it classifies it as 0, so correct, do nothing



- Example 2  $I = \langle 1 \ 0 \ 1 \rangle$  label=1  $W = \langle 0, 0, 0 \rangle$
- Perceptron ( $1 \times 0 + 0 \times 0 + 1 \times 0 = 0$ ) output  $\rightarrow 0$   
 $\rightarrow$  it classifies it as **0, while it should be 1, so we add input to weights**  
 $W = \langle 0, 0, 0 \rangle + \langle 1, 0, 1 \rangle = \langle 1, 0, 1 \rangle$

# Perceptron: Example

- Example 3  $I = \langle 1 \ 1 \ 0 \rangle$  label=1  $W = \langle 1, 0, 1 \rangle$
- Perceptron  $(1 \times 0 + 1 \times 0 + 0 \times 0 > 0)$  output = 1  
→ it classifies it as 1, correct, do nothing  
 $W = \langle 1, 0, 1 \rangle$
- Example 4  $I = \langle 1 \ 1 \ 1 \rangle$  label=1  $W = \langle 1, 0, 1 \rangle$
- Perceptron  $(1 \times 0 + 1 \times 0 + 1 \times 0 > 0)$  output = 1  
→ it classifies it as 1, correct, do nothing  
 $W = \langle 1, 0, 1 \rangle$



1<sup>st</sup> iteration is completed.  
Repeat until no errors

# Perceptron

---

- Limitations of Perceptron
  - It is too simple, cannot learn complex and effective models
  - It assumes the data can be linearly separatable in the binary classification, but actually it could be non-linear!
    - SVM, we use kernel function to map data to higher-dimension
    - ANN, we can add more layers!!

# Schedule

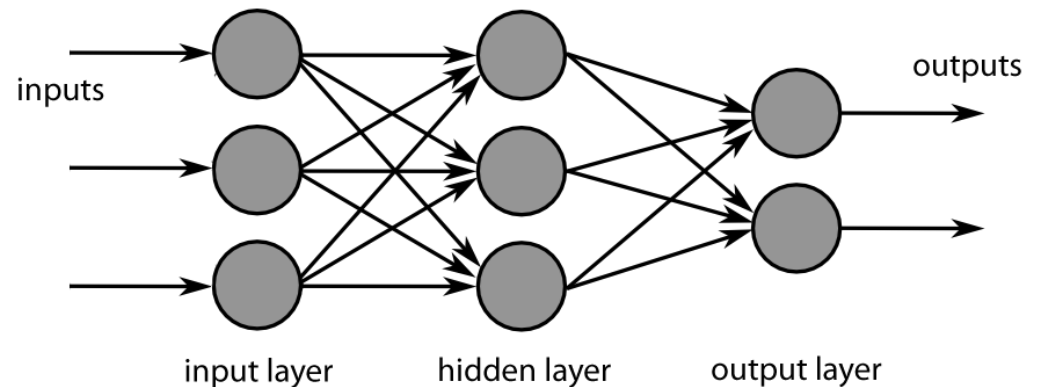
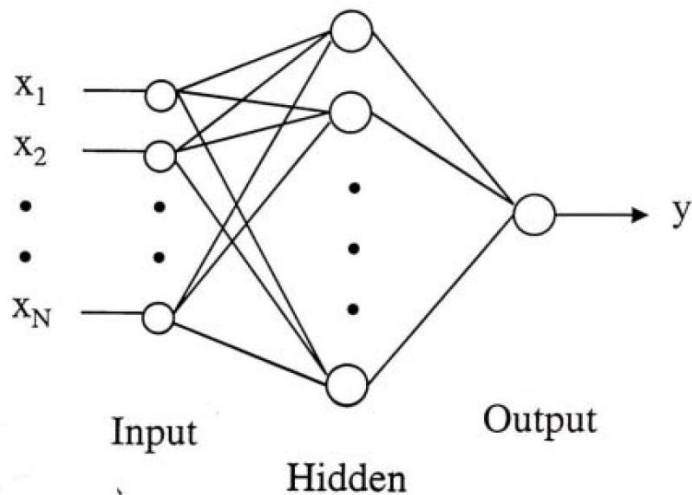
---

- Intro: Neural Networks
- Perceptron and Training
- Multi-layer Feed-forward Networks
- Backpropagation Training
- Neural Networks and Deep Learning



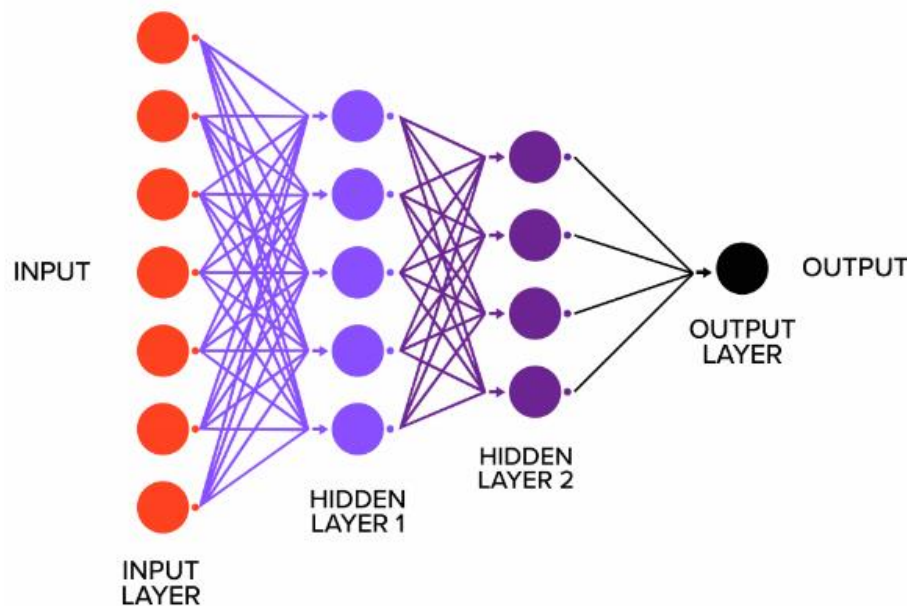
# Multi-layer Feed-forward Networks

- Multi-layer Feed-forward Networks is an extension of the perceptron model. It adds hidden layers to the original perceptron



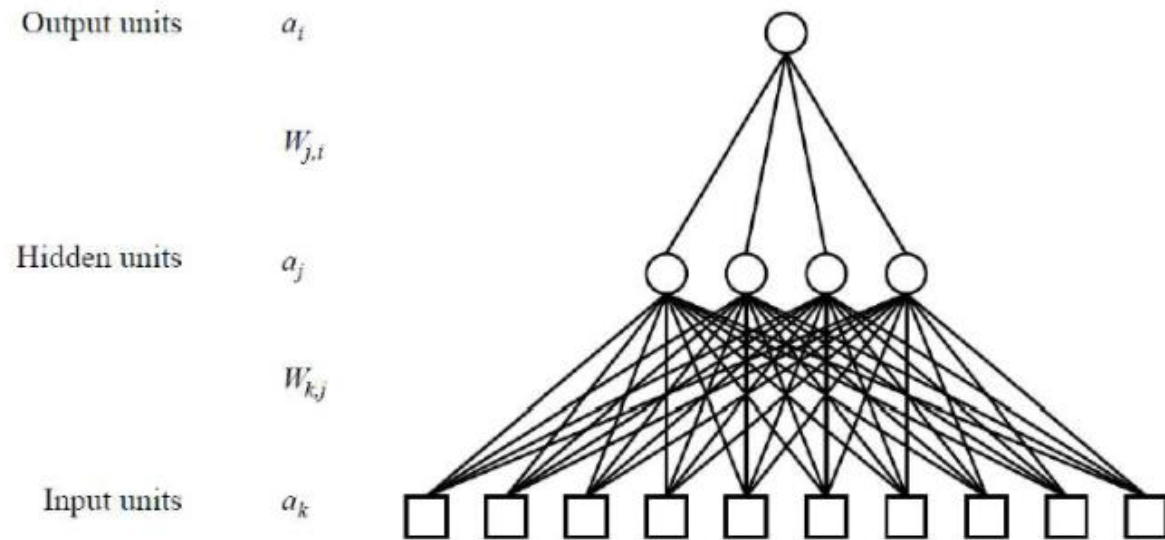
# Multi-layer Feed-forward Networks

- Multi-layer Feed-forward Networks is an extension of the perceptron model. It adds hidden layers to the original perceptron



# Multi-layer Feed-forward Networks

- Input layer: accepts inputs only
- Hidden layers: neurons with functions
- Output layer: produce outputs



$$a_i = g(\sum_j W_{ji} g(\sum_k W_{kj} a_k))$$

# Schedule

---

- Intro: Neural Networks
- Perceptron and Training
- Multi-layer Feed-forward Networks
- Backpropagation Training
- Neural Networks and Deep Learning

# Multi-layer Feed-forward Networks

---

- There are several network structure in neural networks, such as feed-forward neural networks and the recurrent neural networks
- Multi-layer Feed-forward Networks used a forward procedure for predictions. But it was trained by using a Backward propagation approach
- These ANNs are also called BP (Backpropagation) Neural Networks

# Training Phrase

---

- The training phrase is a typical process of machine learning and optimization
- We need to
  - Setup a learning objective as loss function
  - Use appropriate optimizer to learn the parameters
  - It is usually a process of iterative learning

# Loss Function

- The loss function  $L(x, y, y')$  is defined as the amount of utility lost by predicting  $h(x) = y'$  when the correct answer is  $f(x) = y$
- Often a simplified version is used,  $L(y, y')$ , that is independent of  $x$
- Three commonly used loss functions:
  - Absolute value loss:  $L_1(y, y') = |y - y'|$
  - Squared error loss:  $L_2(y, y') = (y - y')^2$
  - 0/1 loss:  $L_{0/1}(y, y') = 0$  if  $y = y'$ , else 1
- Let  $E$  be the set of examples. Total loss  $L(E) = \sum_{e \in E} L(e)$

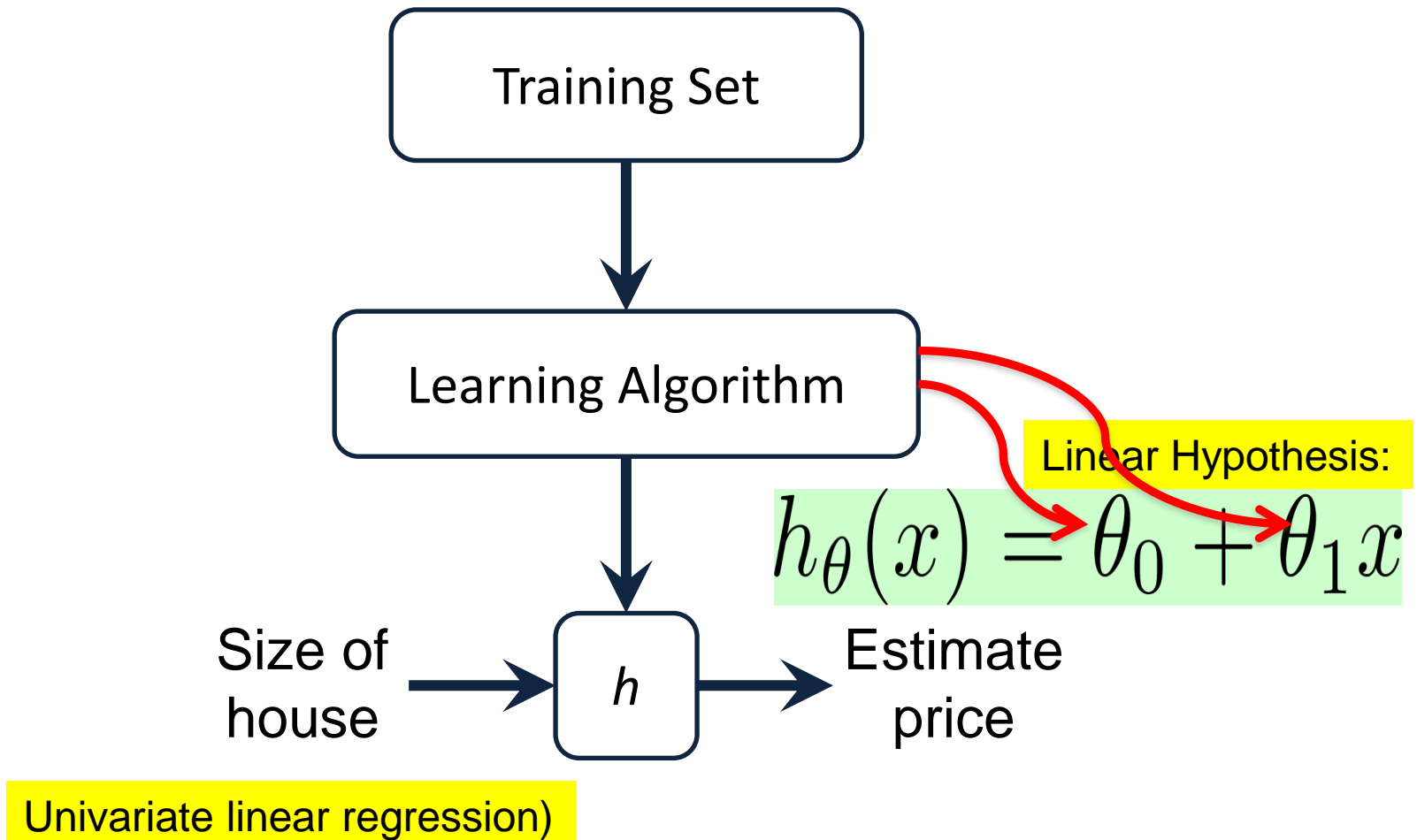
# Optimizer: Gradient Descent

---

- Gradient Descent is widely used as one of the popular optimizers in machine learning, especially in the ANN learning
- Let's use linear regression as one example



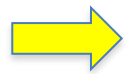
# Example: Linear Regression



# Optimization In Linear Regression

---

- How to apply gradient descent to minimize the cost function for regression



1. a closer look at the cost function
2. applying gradient descent to find the minimum of the cost function

Hypothesis:

$$h_{\theta}(x) = \theta_0 + \theta_1 x$$

Parameters:

$$\theta_0, \theta_1$$

Cost Function:



Sum of squared errors

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

Goal: minimize  $J(\theta_0, \theta_1)$   
 $\theta_0, \theta_1$

- The loss function  $L(x, y, y')$  is defined as the amount of utility lost by predicting  $h(x) = y'$  when the correct answer is  $f(x) = y$
- Often a simplified version is used,  $L(y, y')$ , that is independent of  $x$
- Three commonly used loss functions:
  - Absolute value loss:  $L_1(y, y') = |y - y'|$
  - Squared error loss:  $L_2(y, y') = (y - y')^2$
  - 0/1 loss:  $L_{0/1}(y, y') = 0$  if  $y = y'$ , else 1
- Let  $E$  be the set of examples. Total loss  $L(E) = \sum_{e \in E} L(e)$

# Optimization

---

- There are at least two optimization methods
  - Least square optimization
  - Optimization based on gradient descent

# Least Square Optimization


- Find the optimal point

$$\frac{\partial}{\partial \theta_j} J = 0, J \text{ is the objective function with } \theta_1, \theta_2, \theta_3, \dots$$

- $j = 1, 2, 3, \dots, N+1$ , assume u have N x variables
- Therefore, you will have N+1 functions to be solved
- Drawback: it is complicated if you have many x variables

# Optimization based on Gradient Decent

---

- How to apply gradient descent to minimize the cost function for regression
  1. a closer look at the cost function
  -  2. applying gradient descent to find the minimum of the cost function

# Optimization based on Gradient Decent

---

Have some function  $J(\theta_0, \theta_1)$

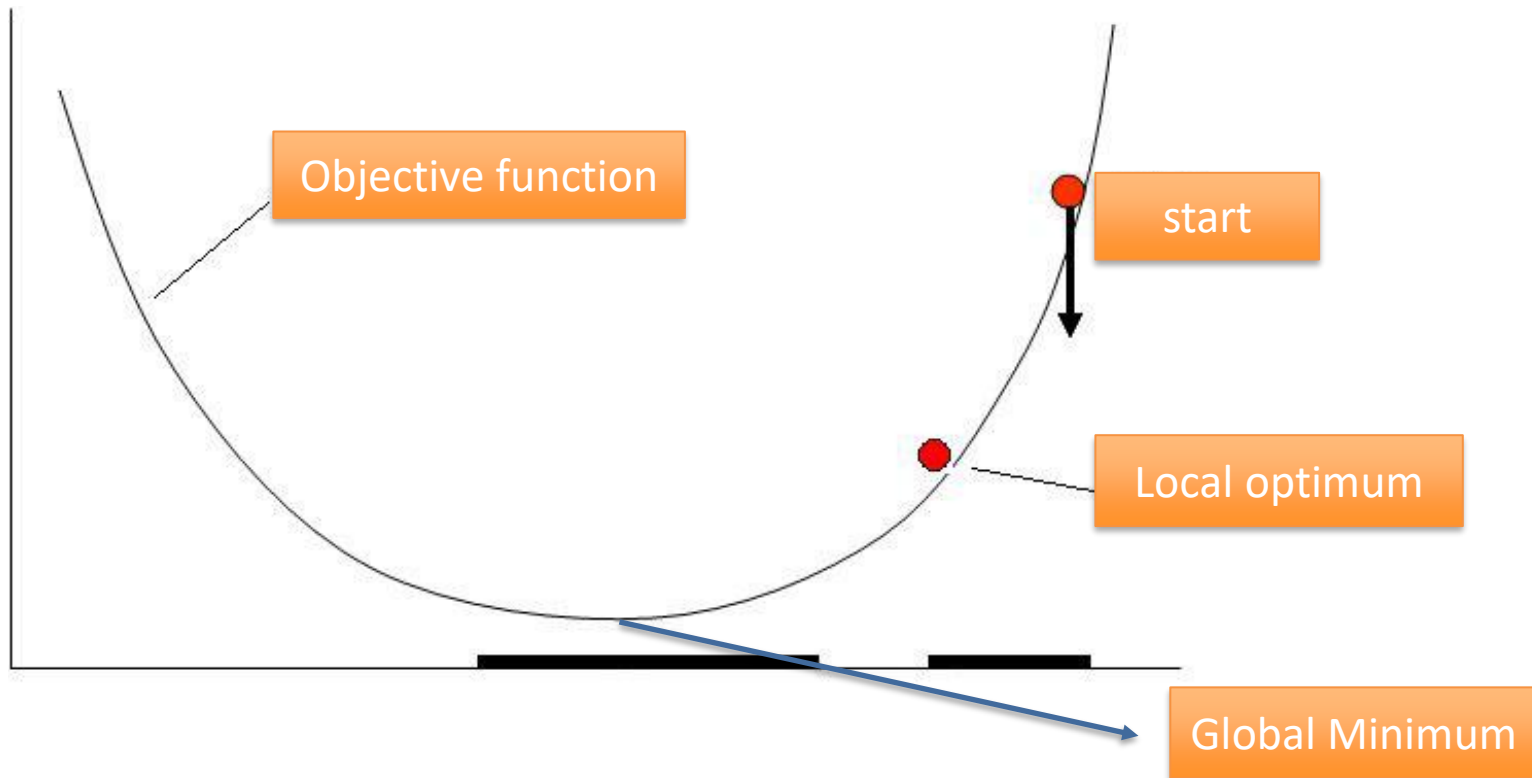
Want  $\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$

## **Gradient descent algorithm outline:**

- Start with some  $\theta_0, \theta_1$
- Keep changing  $\theta_0, \theta_1$  to reduce  $J(\theta_0, \theta_1)$   
until we hopefully end up at a minimum

# Gradient Descent

- Find the optimal point





# Optimization based on Gradient Decent

Have some function  $J(\theta_0, \theta_1)$

Want  $\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$

## Gradient descent algorithm

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad \text{(simultaneously update } j = 0 \text{ and } j = 1)$$

}



1<sup>st</sup> Derivative

# Optimization based on Gradient Decent

Have some function  $J(\theta_0, \theta_1)$

Want  $\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$

## Gradient descent algorithm

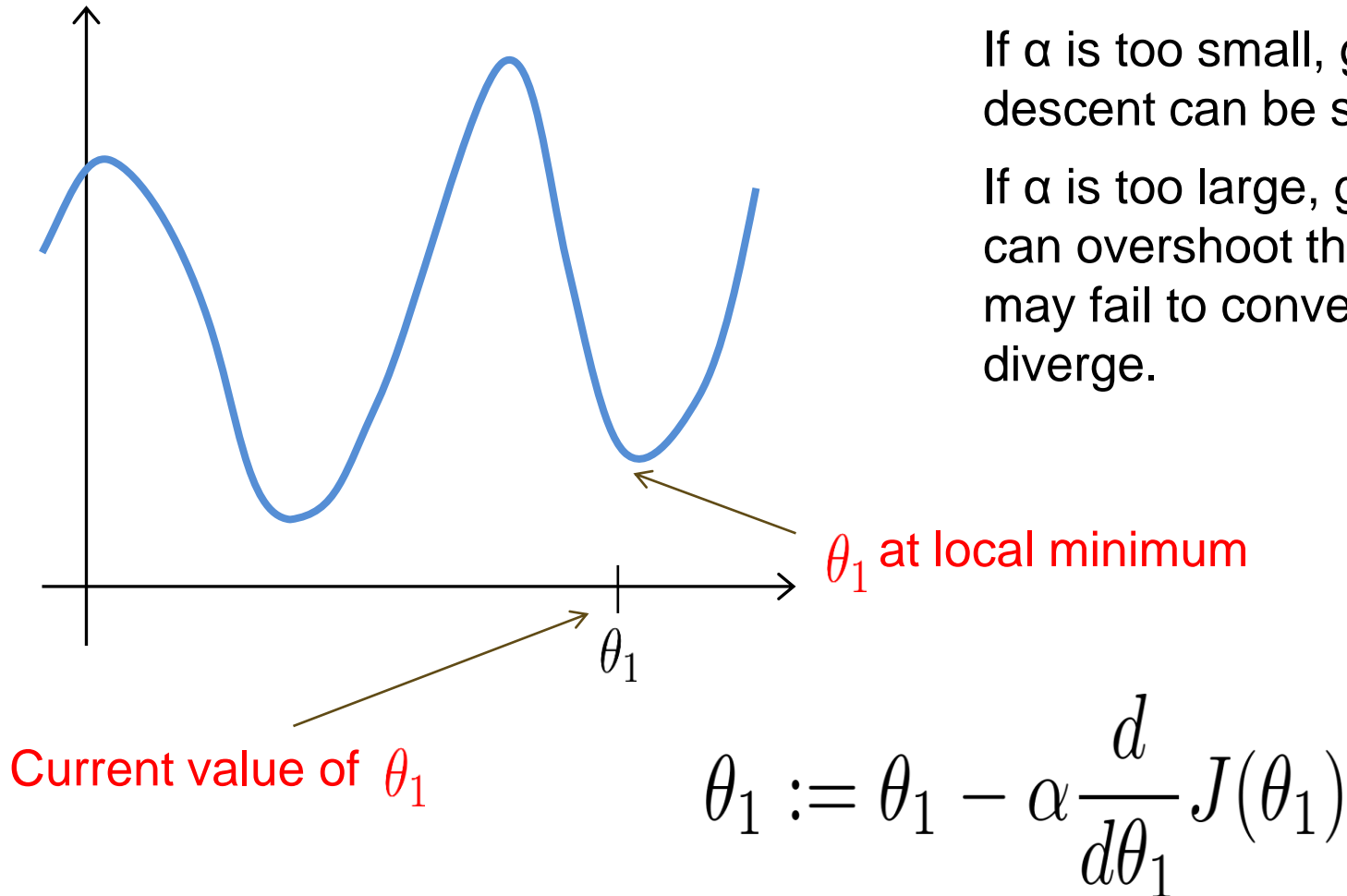
repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad (\text{simultaneously update } j = 0 \text{ and } j = 1)$$

}

learning rate

# Optimization based on Gradient Decent



If  $\alpha$  is too small, gradient descent can be slow.

If  $\alpha$  is too large, gradient descent can overshoot the minimum. It may fail to converge, or even diverge.

# Optimization based on Gradient Decent

---

Gradient descent can converge to a local minimum, even with the learning rate  $\alpha$  fixed.

$$\theta_1 := \theta_1 - \alpha \frac{d}{d\theta_1} J(\theta_1)$$

## Gradient descent algorithm

repeat until convergence {  
     $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$   
    (for  $j = 1$  and  $j = 0$ )  
}

## Linear Regression Model

$$h_{\theta}(x) = \theta_0 + \theta_1 x$$

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

# Gradient descent algorithm

repeat until convergence {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$$

} **update  
 $\theta_0$  and  $\theta_1$   
simultaneously**

}

# Backpropagation Learning

## ANN needs a process of weight training

- A set of examples, each with input vector  $x$  and output vector  $y$
- Squared error loss:  $Loss = \sum_k Loss_k$ ,  $Loss_k = (y_k - a_k)^2$ , where  $a_k$  is the  $k$ -th output of the neural net
- The weights are adjusted as follows:  
$$w_{ij} \leftarrow w_{ij} - \alpha \partial Loss / \partial w_{ij}$$
- How can we compute the gradient efficiently given an arbitrary network structure?
- Answer: backpropagation algorithm

# Forward vs Backward in ANN

Forward phase:

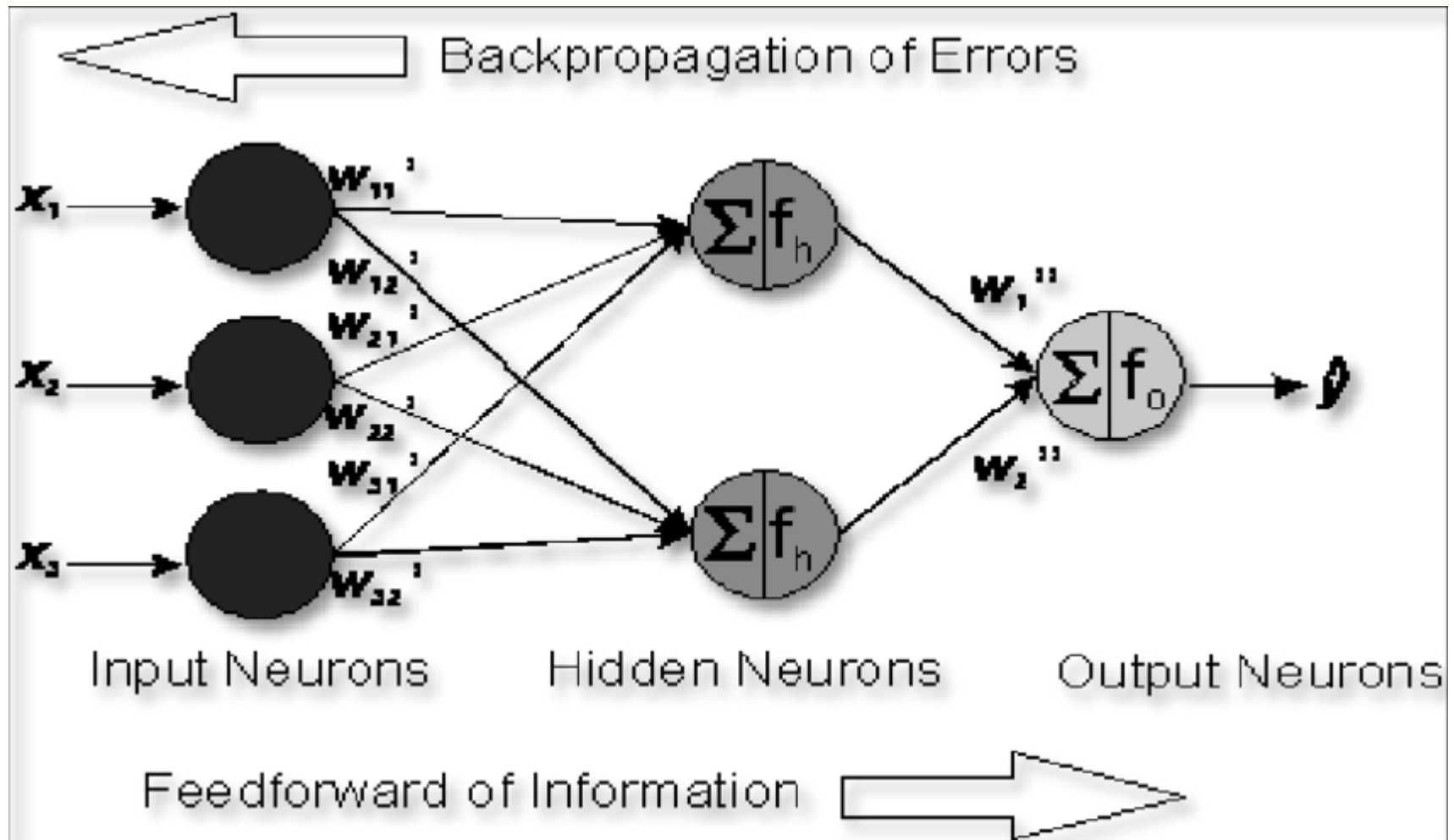
- Propagate inputs forward to compute the output of each unit
- Output  $a_j$  at unit  $j$ :  $a_j = g(in_j)$  where  $in_j = \sum_i w_{ij}a_i$

Backward phase:

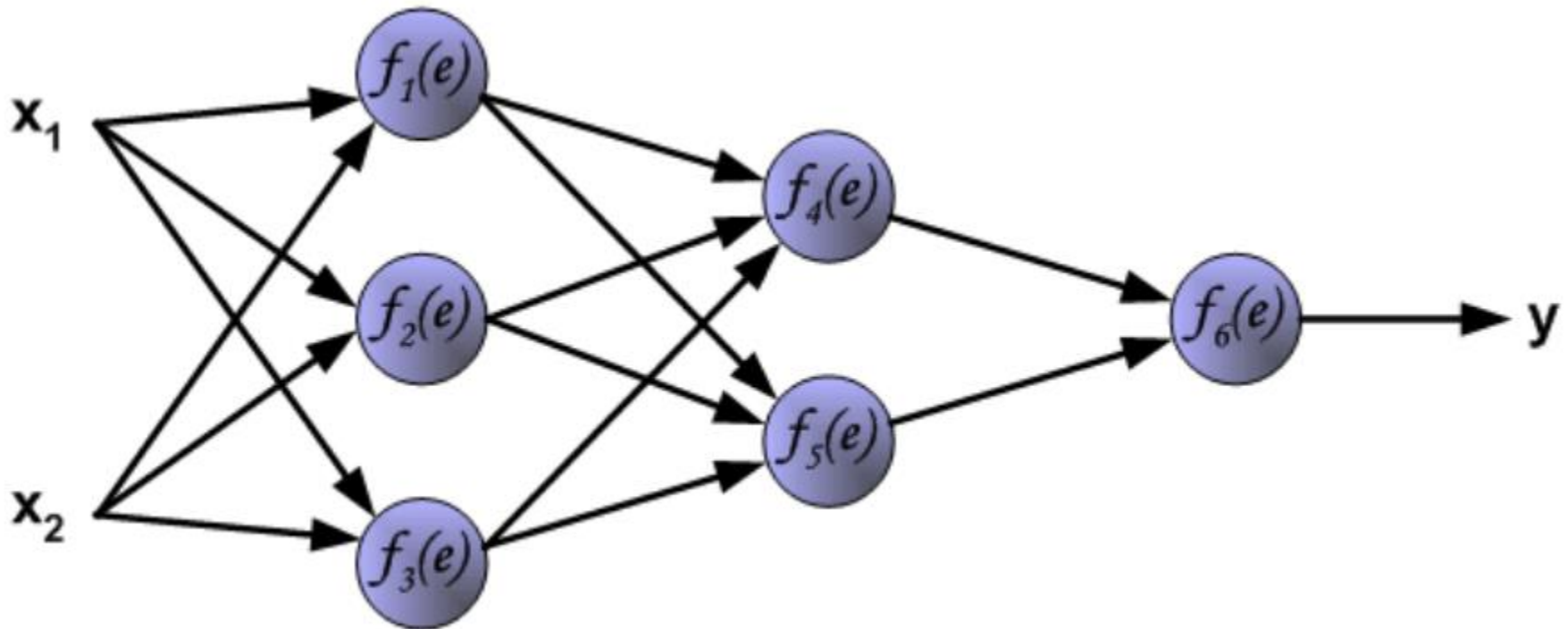
- Propagate errors backward
- For an output unit  $j$ :  $\Delta_j = g'(in_j)(y_j - a_j)$
- For an hidden unit  $i$ :  $\Delta_i = g'(in_i) \sum_j w_{ij} \Delta_j$



# Forward vs Backward in ANN

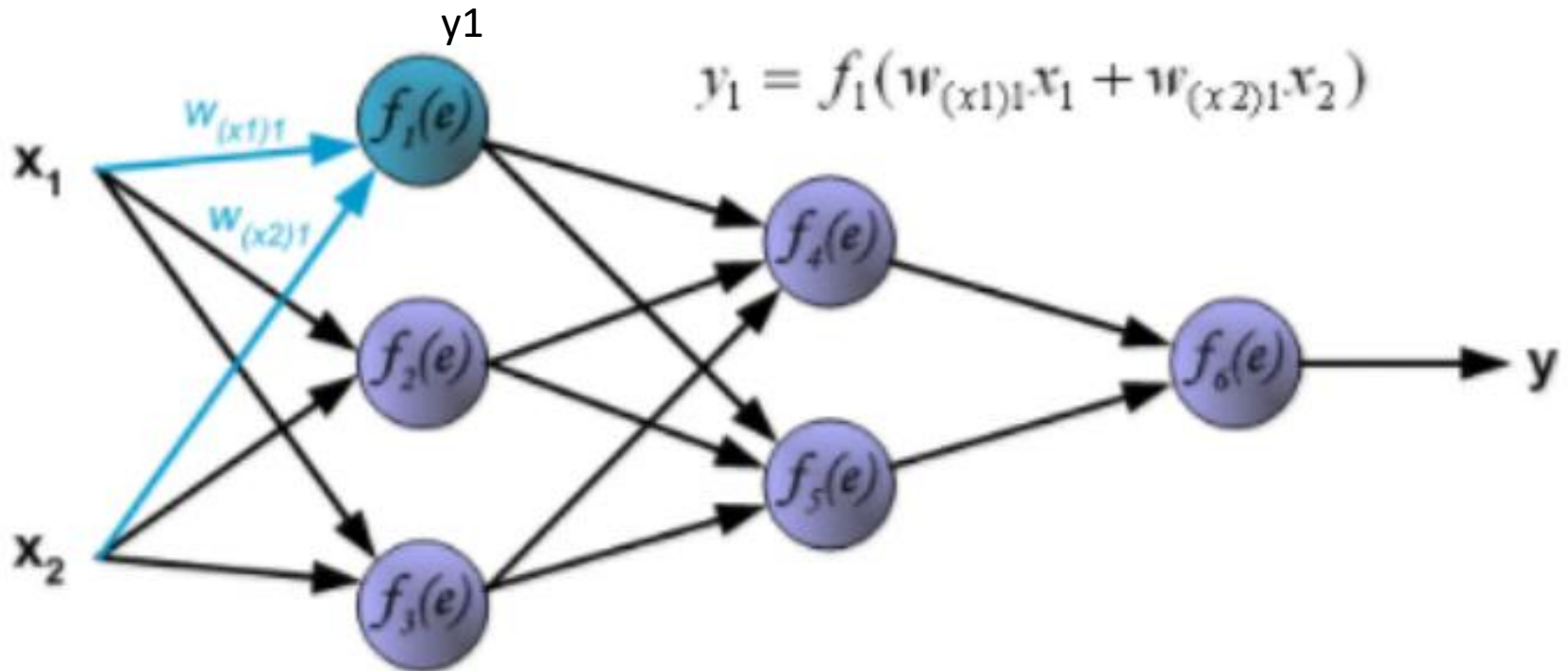


# Example



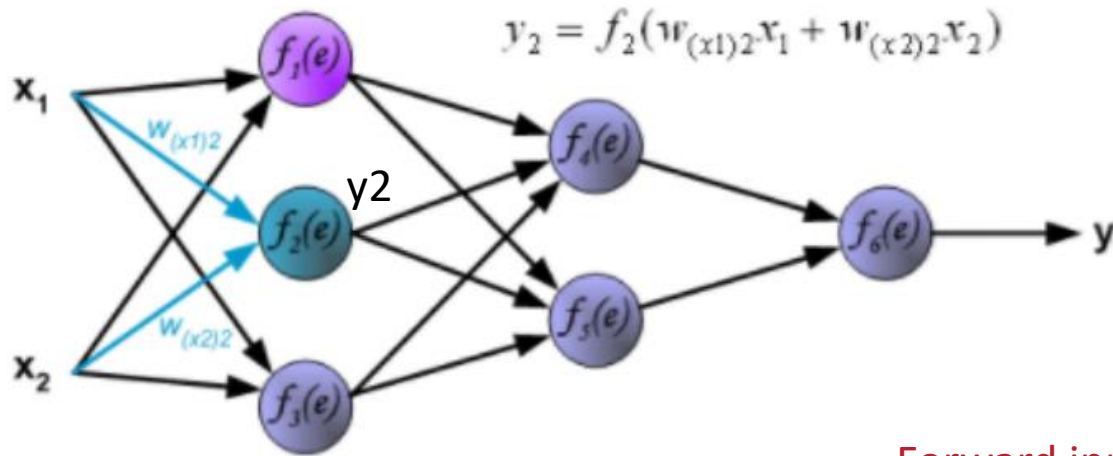
f functions are used to denote the activation functions

# Forward in ANN

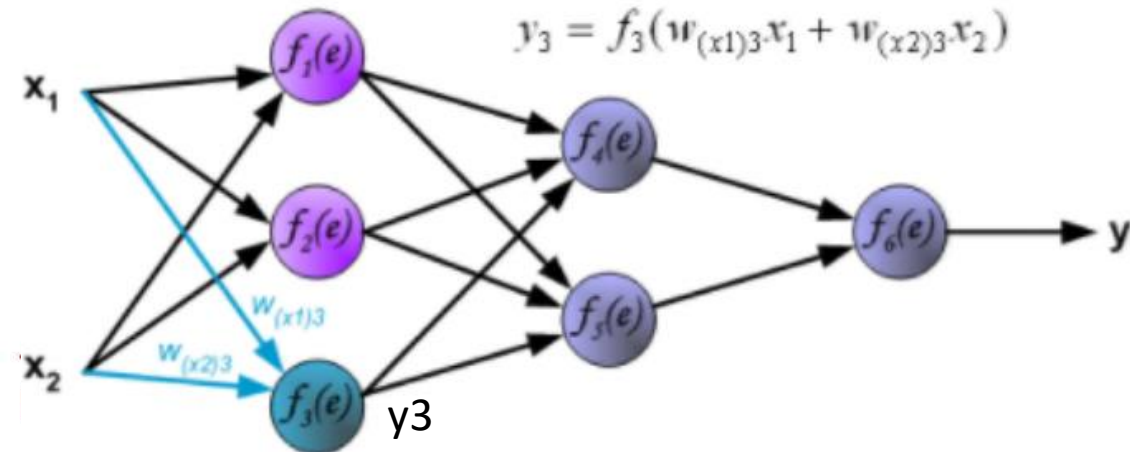


Forward inputs to the nodes at hidden layers

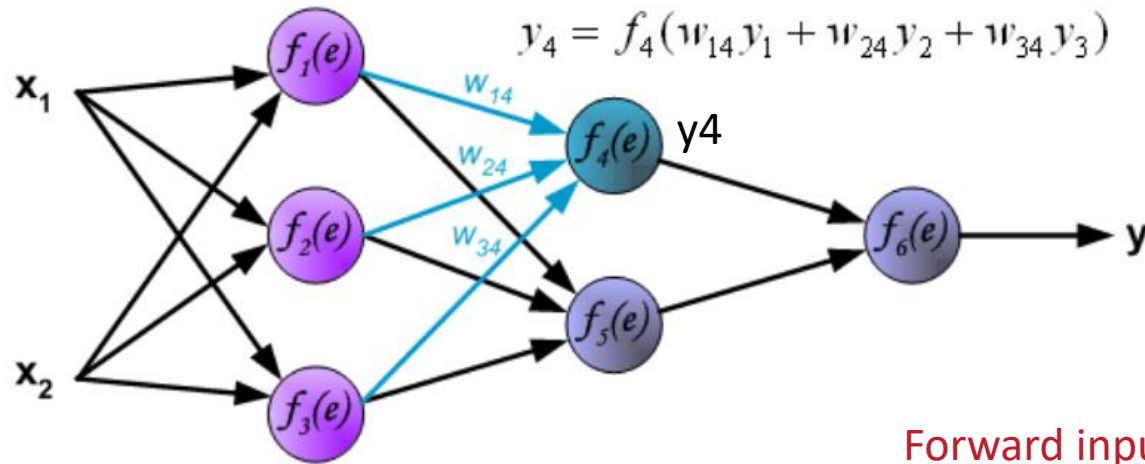
# Forward in ANN



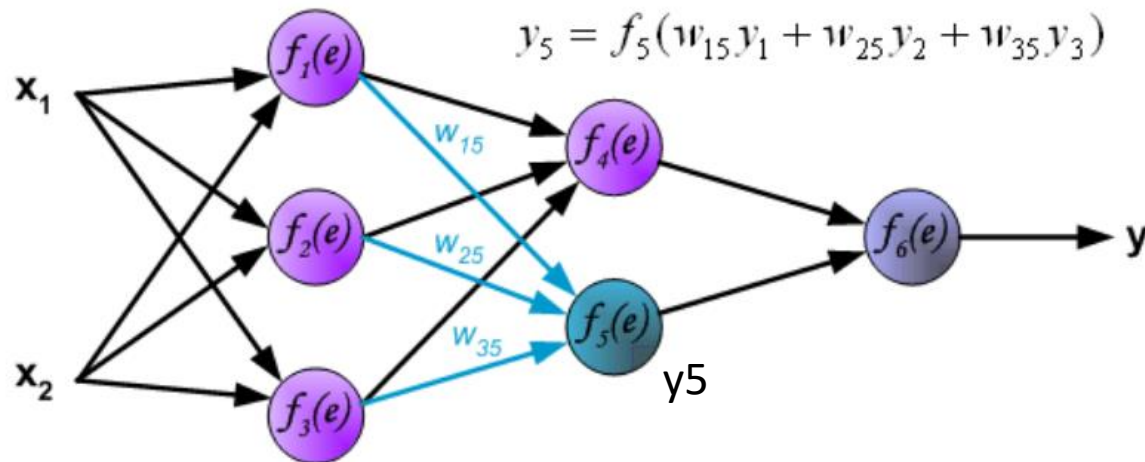
Forward inputs to the nodes at hidden layers



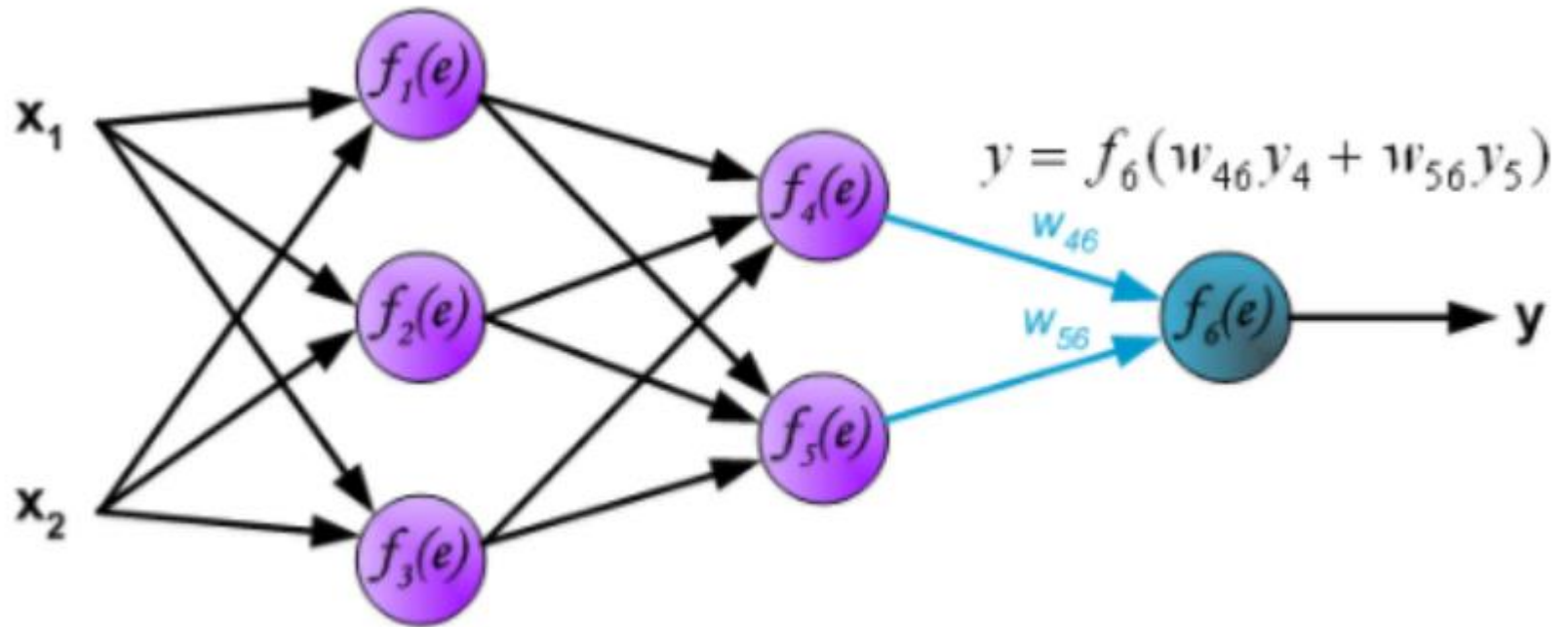
# Forward in ANN



Forward inputs to the nodes at hidden layers

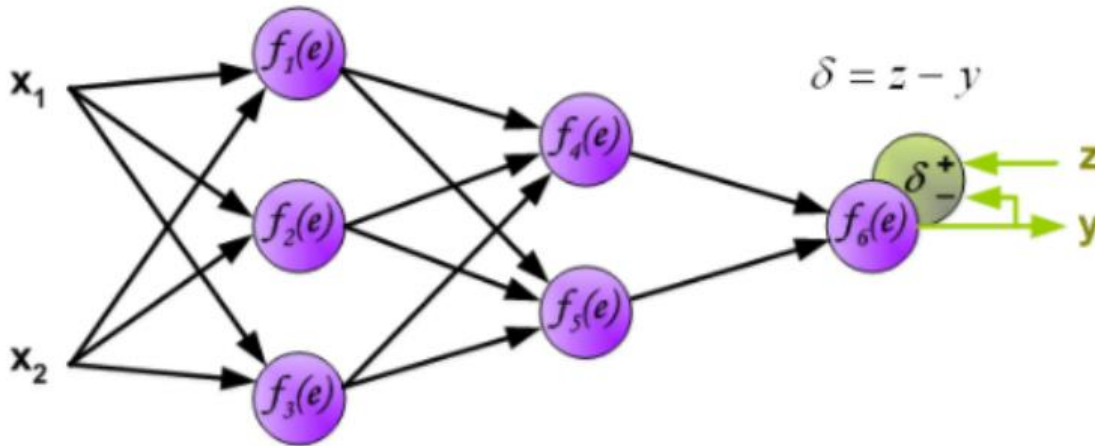


# Forward in ANN

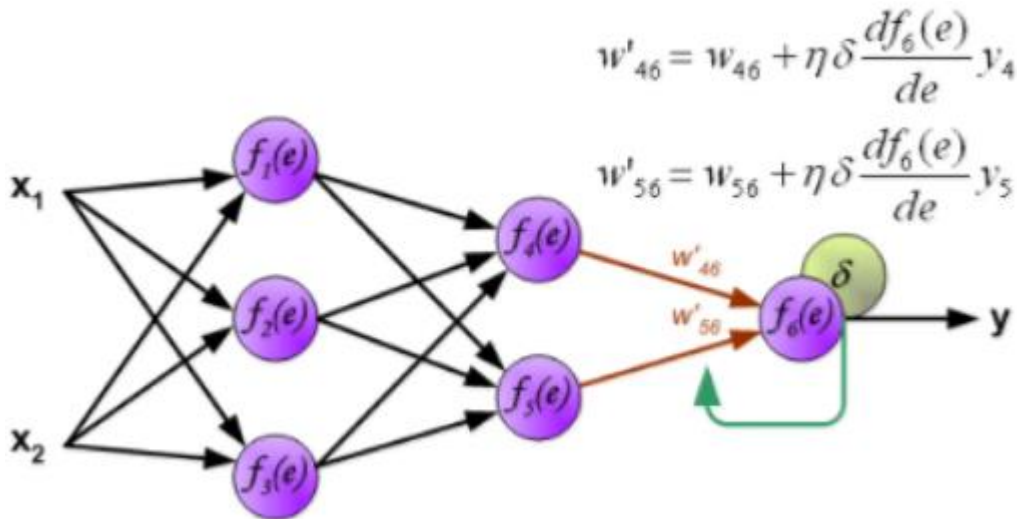


Forward inputs to the last node to produce the predictions

# Backward in ANN



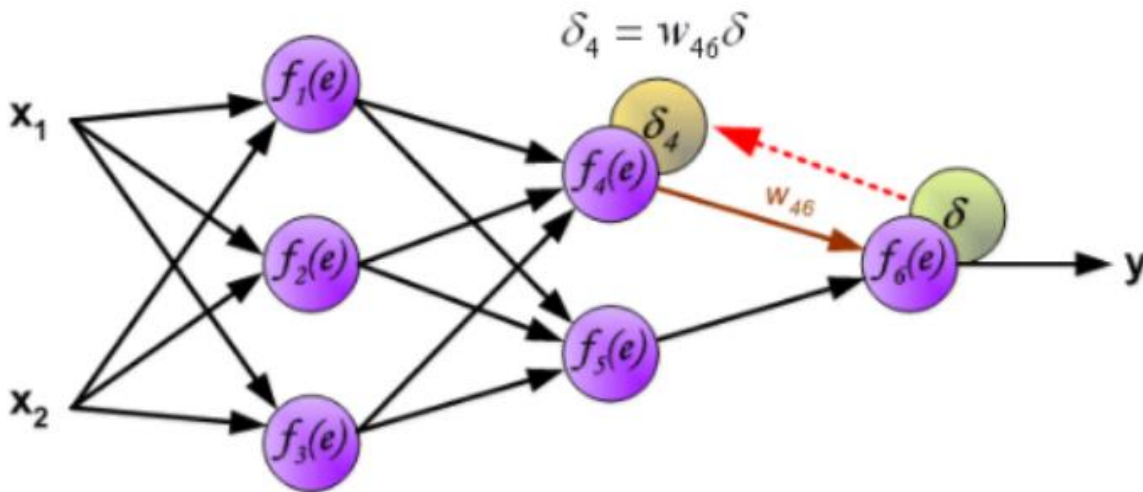
Backward the errors



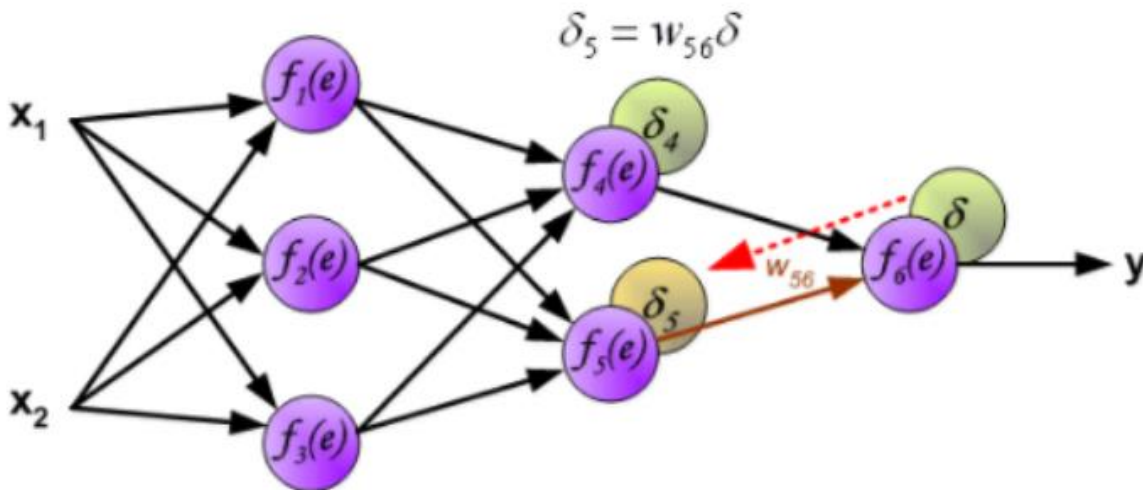
Update weights



# Backward in ANN



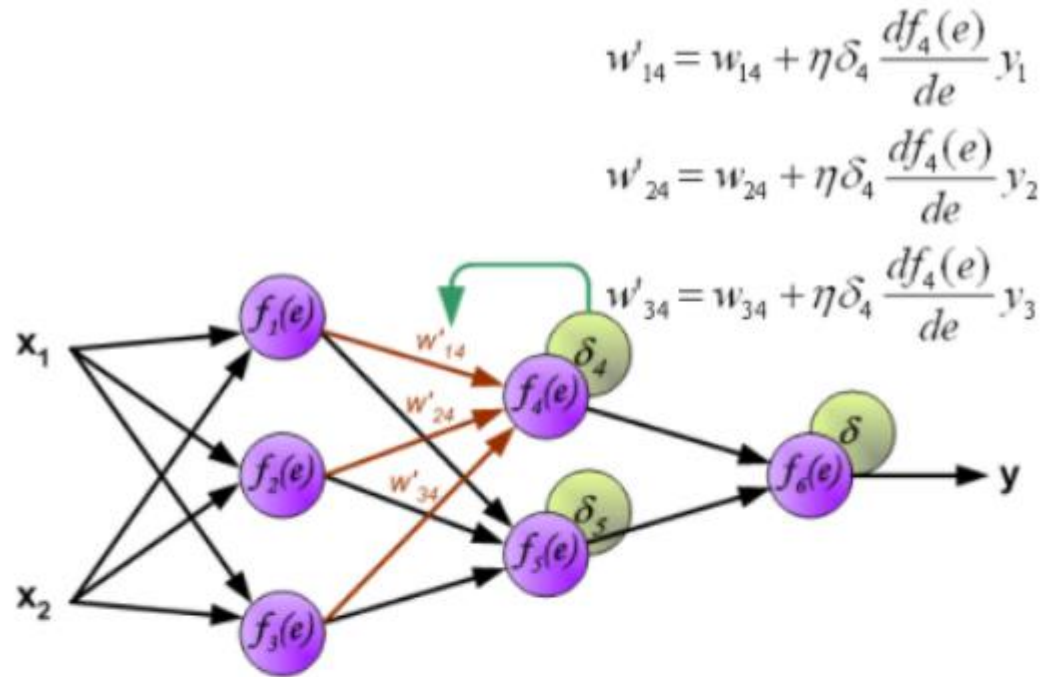
Backward the errors



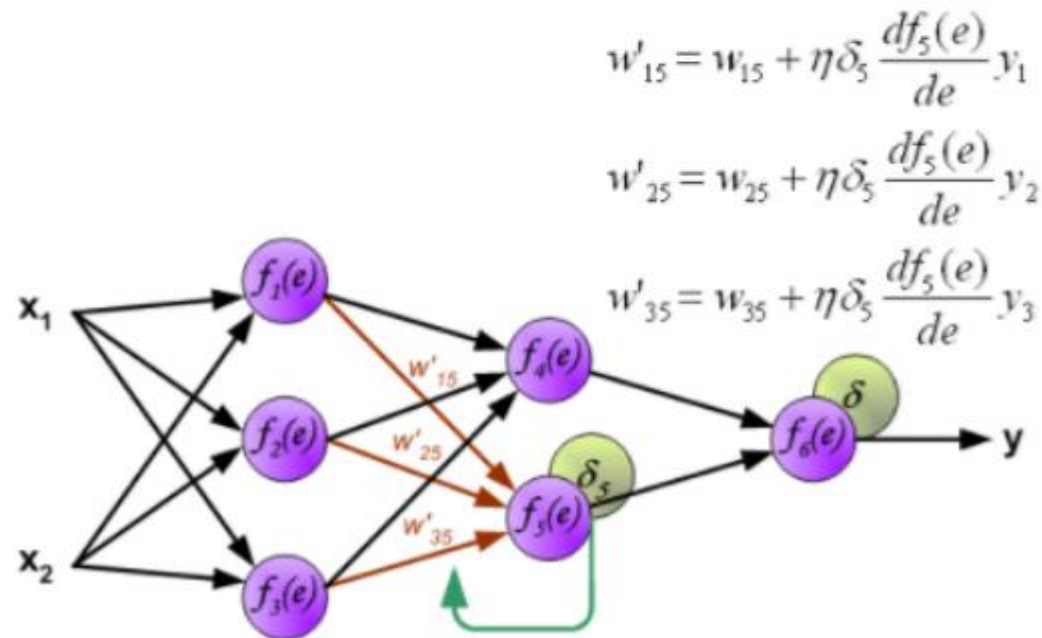
Backward the errors



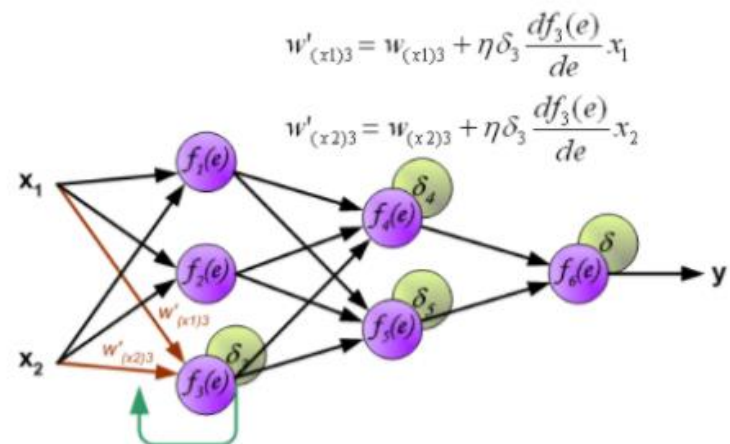
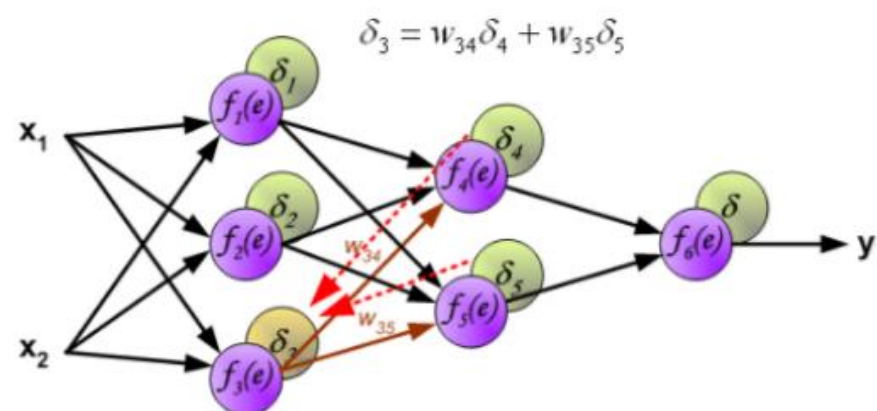
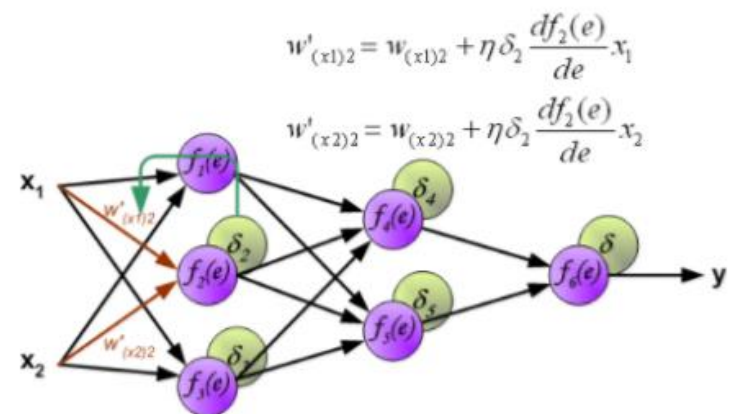
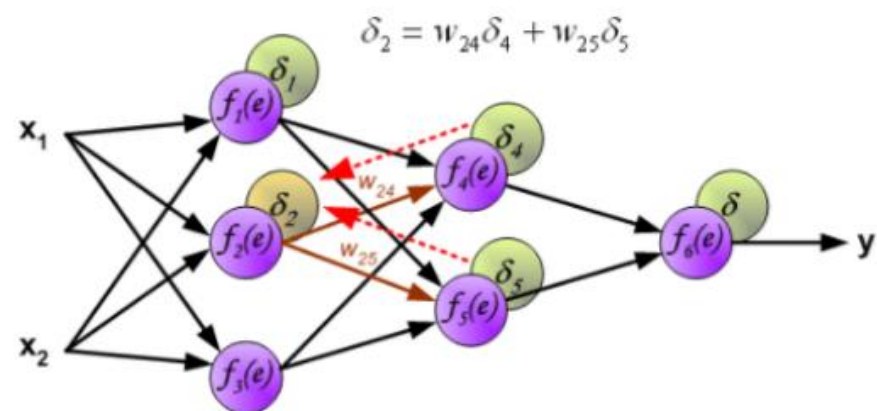
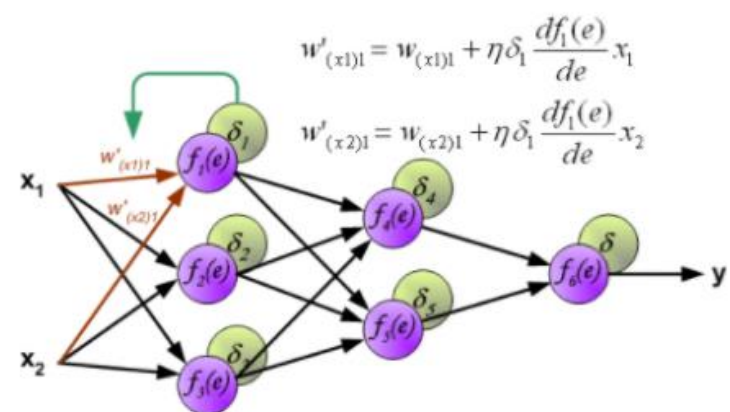
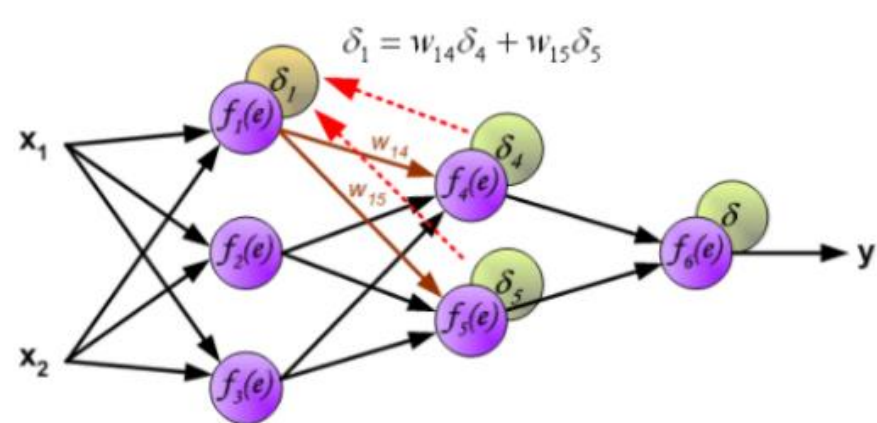
# Backward in ANN



Update weights



Update weights



# Backpropagation Learning

```
function BACK-PROP-LEARNING(examples, network) returns a neural network
  inputs: examples, a set of examples, each with input vector  $\mathbf{x}$  and output vector  $\mathbf{y}$ 
         network, a multilayer network with  $L$  layers, weights  $w_{i,j}$ , activation function  $g$ 
  local variables:  $\Delta$ , a vector of errors, indexed by network node

  repeat
    for each weight  $w_{i,j}$  in network do
       $w_{i,j} \leftarrow$  a small random number
    for each example  $(\mathbf{x}, \mathbf{y})$  in examples do
      /* Propagate the inputs forward to compute the outputs */
      for each node  $i$  in the input layer do
         $a_i \leftarrow x_i$ 
      for  $\ell = 2$  to  $L$  do
        for each node  $j$  in layer  $\ell$  do
           $in_j \leftarrow \sum_i w_{i,j} a_i$ 
           $a_j \leftarrow g(in_j)$ 
      /* Propagate deltas backward from output layer to input layer */
      for each node  $j$  in the output layer do
         $\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$ 
      for  $\ell = L - 1$  to  $1$  do
        for each node  $i$  in layer  $\ell$  do
           $\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j} \Delta[j]$ 
      /* Update every weight in network using deltas */
      for each weight  $w_{i,j}$  in network do
         $w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$ 
  until some stopping criterion is satisfied
  return network
```

# Backpropagation Learning

```

function BACK-PROP-LEARNING(examples, network) returns a neural network
  inputs: examples, a set of examples, each with input vector  $\mathbf{x}$  and output vector  $\mathbf{y}$ 
         network, a multilayer network with  $L$  layers, weights  $w_{i,j}$ , activation function  $g$ 
  local variables:  $\Delta$ , a vector of errors, indexed by network node

  repeat
    for each weight  $w_{i,j}$  in network do
       $w_{i,j} \leftarrow$  a small random number
    for each example  $(\mathbf{x}, \mathbf{y})$  in examples do
      /* Propagate the inputs forward to compute the outputs */
      for each node  $i$  in the input layer do
         $a_i \leftarrow x_i$ 
      for  $\ell = 2$  to  $L$  do
        for each node  $j$  in layer  $\ell$  do
           $in_j \leftarrow \sum_i w_{i,j} a_i$ 
           $a_j \leftarrow g(in_j)$ 
      /* Propagate deltas backward from output layer to input layer */
      for each node  $j$  in the output layer do
         $\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$ 
      for  $\ell = L - 1$  to  $1$  do
        for each node  $i$  in layer  $\ell$  do
           $\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j} \Delta[j]$ 
      /* Update every weight in network using deltas */
      for each weight  $w_{i,j}$  in network do
         $w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$ 
  until some stopping criterion is satisfied
  return network
  
```

Initialization

Forward Phrase

Backward Phrase

Iterative  
Learning



# Schedule

---

- Intro: Neural Networks
- Perceptron and Training
- Multi-layer Feed-forward Networks
- Backpropagation Training
- Neural Networks and Deep Learning

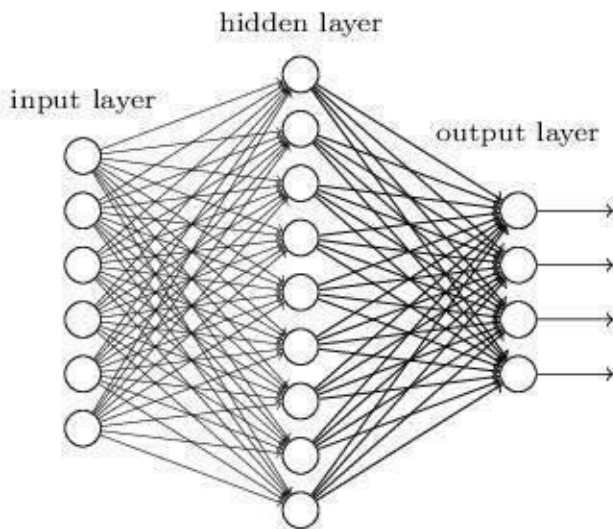
# ANN vs Deep Learning

---

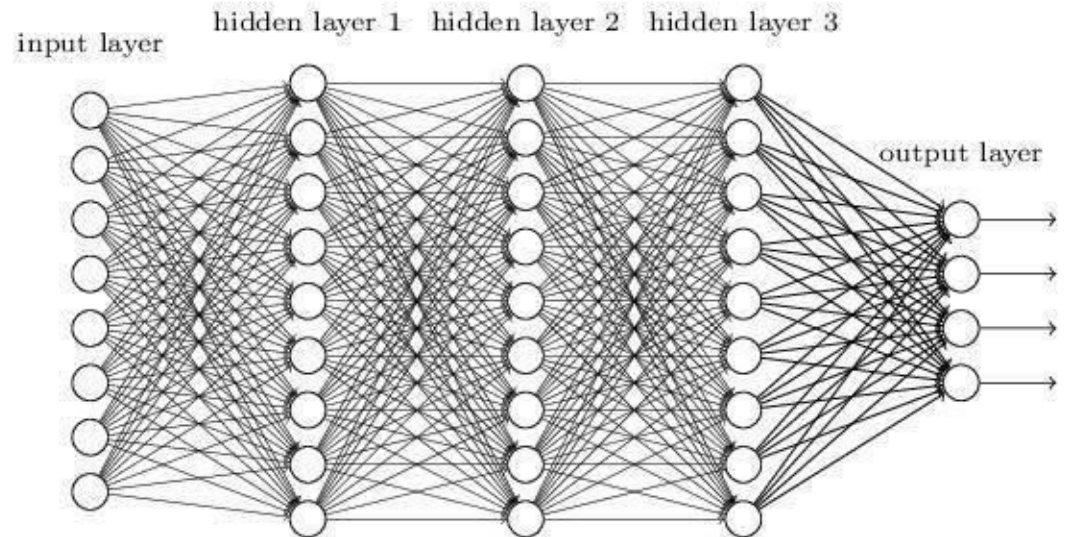
- To make ANN more powerful, there are two solutions
  - Add more neurons in the hidden layer
  - Add more hidden layers
- Deep Learning
  - Traditional ANN only has 3 layers. Deep learning utilizes neural networks with multiple layers
  - Deep learning have more structures for neural networks, such as ANN, Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), and so forth
  - Deep learning is not related to neural networks only. It also correlates with computing, such as GPU

# ANN vs Deep Learning

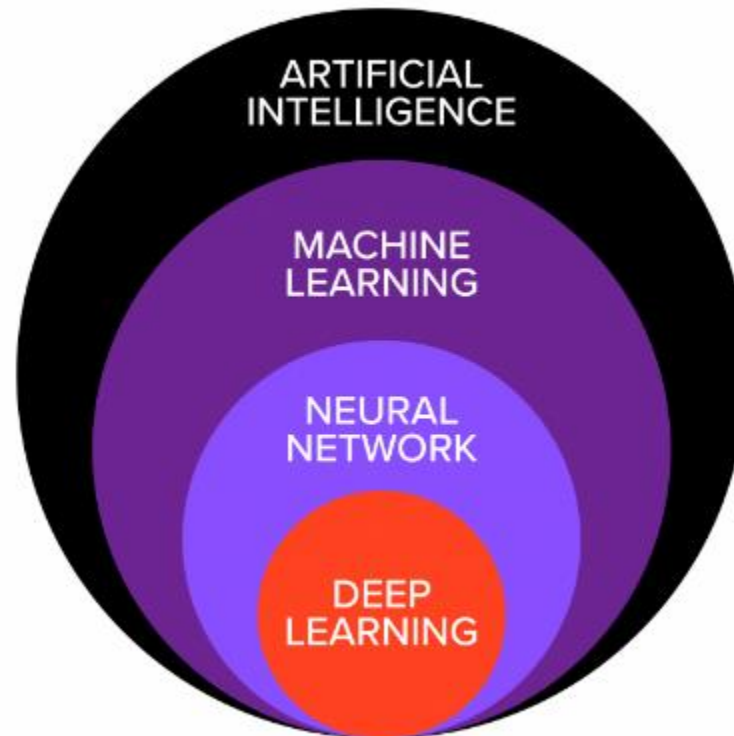
"Non-deep" feedforward neural network



Deep neural network



# ANN vs Deep Learning





# Final Notes

---

- Neural networks are able to learn good feature representations in order to better capture non-linear relationship.
- Neural networks are also considered as one of the techniques for feature engineering – a process of unsupervised learning
- Once the features are learnt or represented, we can utilize these features for predictive models, e.g., classifications or regressions. We will discuss related python coding next week