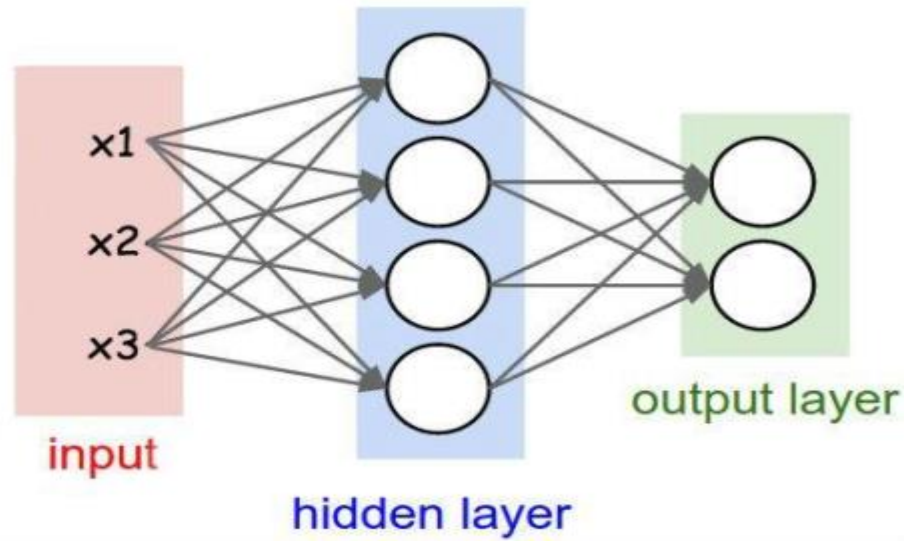
# Multilayer perceptrons

# Why do we need neural network?

- Perceptrons only perform linear classification: The perceptron cannot classify non-linearly separable data since it can only create linear boundaries while training.

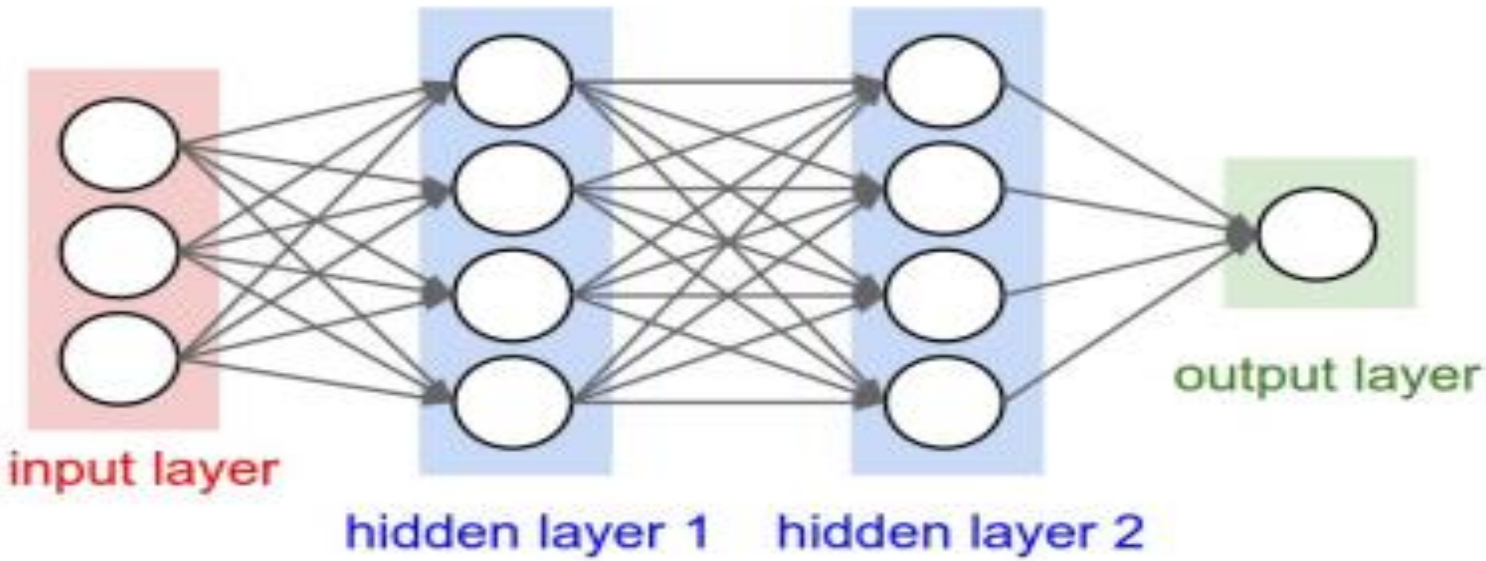
Solution: Use multiple layers of perceptrons but with a differentiable non-linearity activation function.

# Feed forward neural network




- Each unit computes its value based on linear combination of values of units that point into it, and an activation function
- Naming conventions: a 2-layer neural network. One layer of hidden units, One output layer (we do not count the inputs as a layer)

# Feed forward neural network




- A 3-layer neural network with two layers of hidden units
- Naming conventions: a N-layer neural network: N - 1 layers of hidden units, One output layer

# Why do we need more layers?

- Individual features may represent local patterns in data
- Complex patterns: combinations of local patterns
- Options:
  - A large number of perceptrons to learn every possible complex pattern (potentially exponential number of patterns) -- OR
  - A much smaller hierarchial network that builds complex patterns from local patterns (much much more efficient)

# How can we use neural networks?

- Neural networks can be used to detect pattern in both classification and regression problems.

- Neural networks can also be used to extract features in a given unlabeled dataset as well.

# Learning in neural networks

# Learning in neural network

Learning some pattern in a neural network is equal to finding the right weight adjustments of a perceptron that minimize the objective function. The weights represent the learning in a neural network.

# Back propagation: Intutition

# Examples in class

# Back propagation in neural network

# Learning in a neural network

- Minimize the objective function:

  E = $\sum_{n=1}^{N} loss(y^{(n)}, o^{(n)})$

  where, loss function could be one of the following:

  Squared Loss = $\frac{1}{2}\sum_k (y_k^{(n)} - o_k^{(n)})^2$

  Cross entropy loss = $-\sum_k y_k^{(n)} \log o_k^{(n)}$

# Learning in a neural network

- Use gradient descent approach to learn weights while minimizing the loss/error E. To implement this procedure we need to calculate the rate of change in error w.r.t weights i.e., error derivative for the weights (EW)

$$w_{ij} = w_{ij} - \eta \frac{\partial E}{\partial w_{ij}}$$

# Error gradients of weight

- *Lets assume that* $\delta_j$ *is the rate of change in error wrt the output of unit j.*

$$\frac{\partial E}{\partial w_{ij}} = \overset{\text{Error gradient}}{\frac{\partial E}{\partial o_j}} \overset{\text{local gradient}}{\frac{\partial o_j}{\partial w_{ij}}}$$

# Local gradients

$$\frac{\partial o_j}{\partial w_{ij}} = \frac{\partial o_j}{\partial a_j} \frac{\partial a_j}{\partial w_{ij}}$$

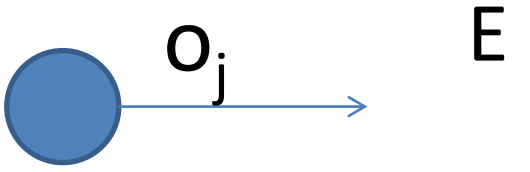
$$= g'(a_j) \frac{\partial \sum_k w_{kj} o_k}{\partial w_{ij}}$$

$$= g'(a_j)\, o_i$$

# Error gradients of output unit

$$\delta_j = \frac{\partial E}{\partial o_j}$$

$$= E'(o_j)$$




*Find* $E'(o_j)$ *for squared loss?*

*Find* $E'(o_j)$ *for Xentropy loss?*

# Error gradients of hidden unit

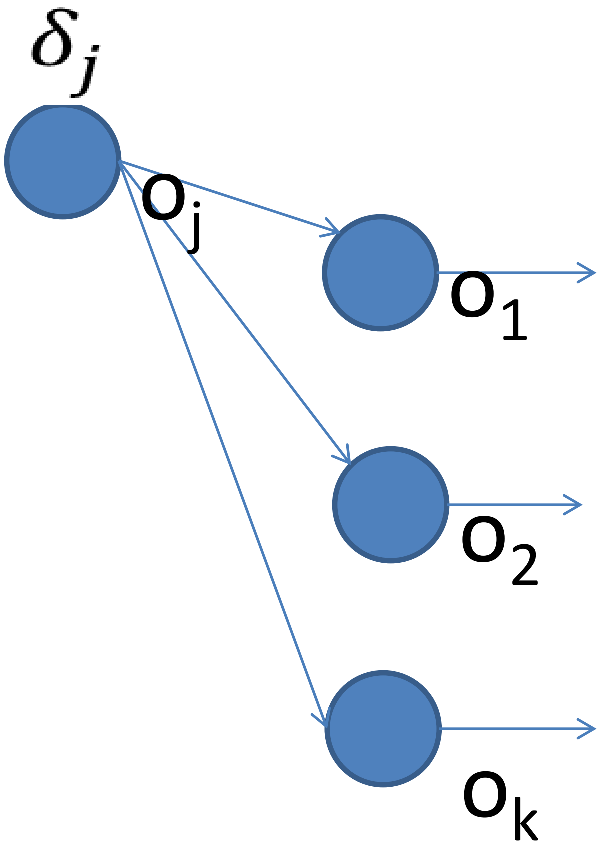
$$\delta_j = \frac{\partial E}{\partial o_j}$$

$$= \sum_k \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial o_j}$$

$$= \sum_k \delta_k \frac{\partial o_k}{\partial o_j}$$

$$= \sum_k \delta_k \frac{\partial \sum_i w_{ik} o_i}{\partial o_j}$$

$$= \sum_k \delta_k w_{jk}$$



# Error gradients of weight

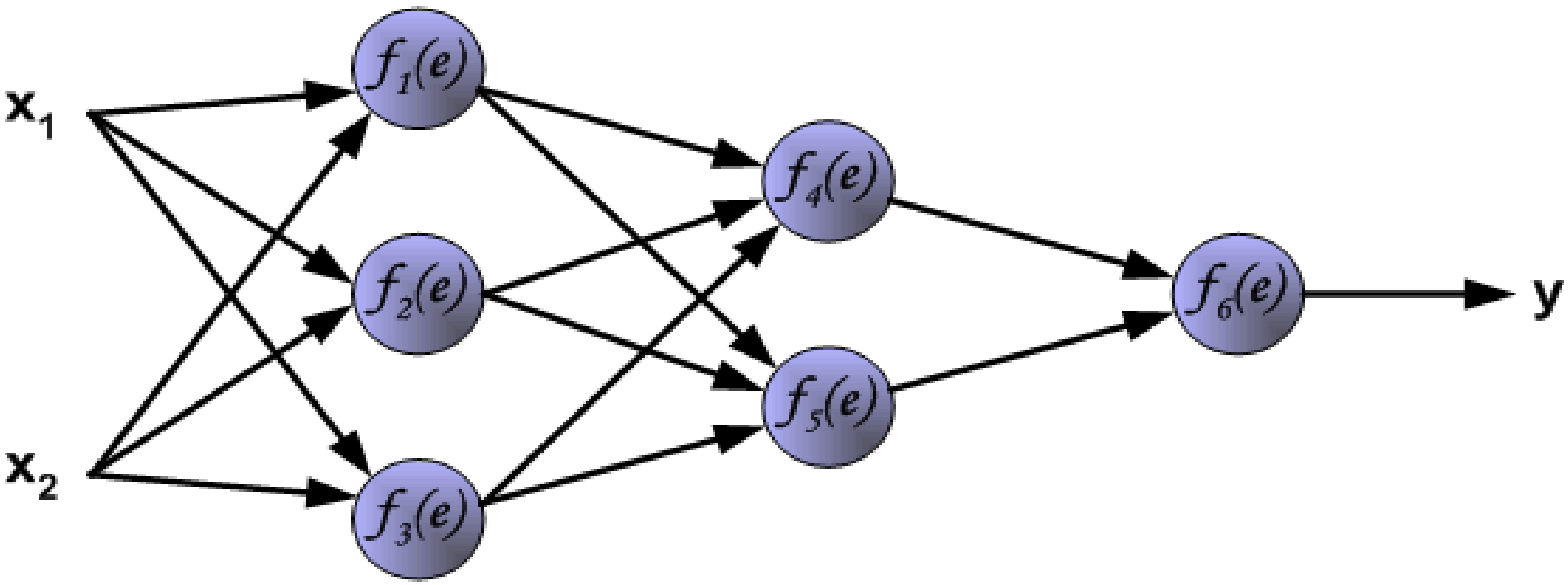
$$\frac{\partial E}{\partial w_{ij}} = \overbrace{\frac{\partial E}{\partial o_j}}^{\text{Error gradient}} \overbrace{\frac{\partial o_j}{\partial w_{ij}}}^{\text{local gradient}}$$

$$= \delta_j \; g'(a_j)\, o_i$$

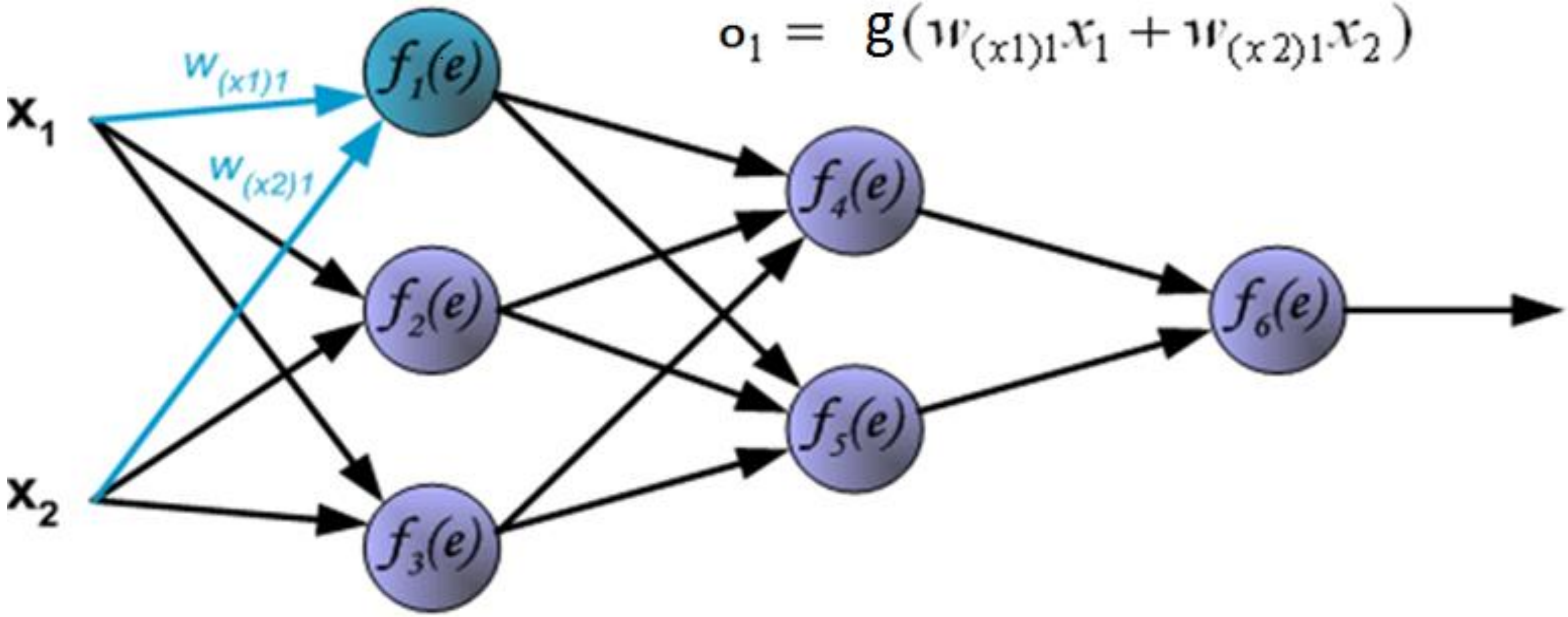
- Calculate $o_i$ using forward step
- Calculate $\delta_j$ using back propagation

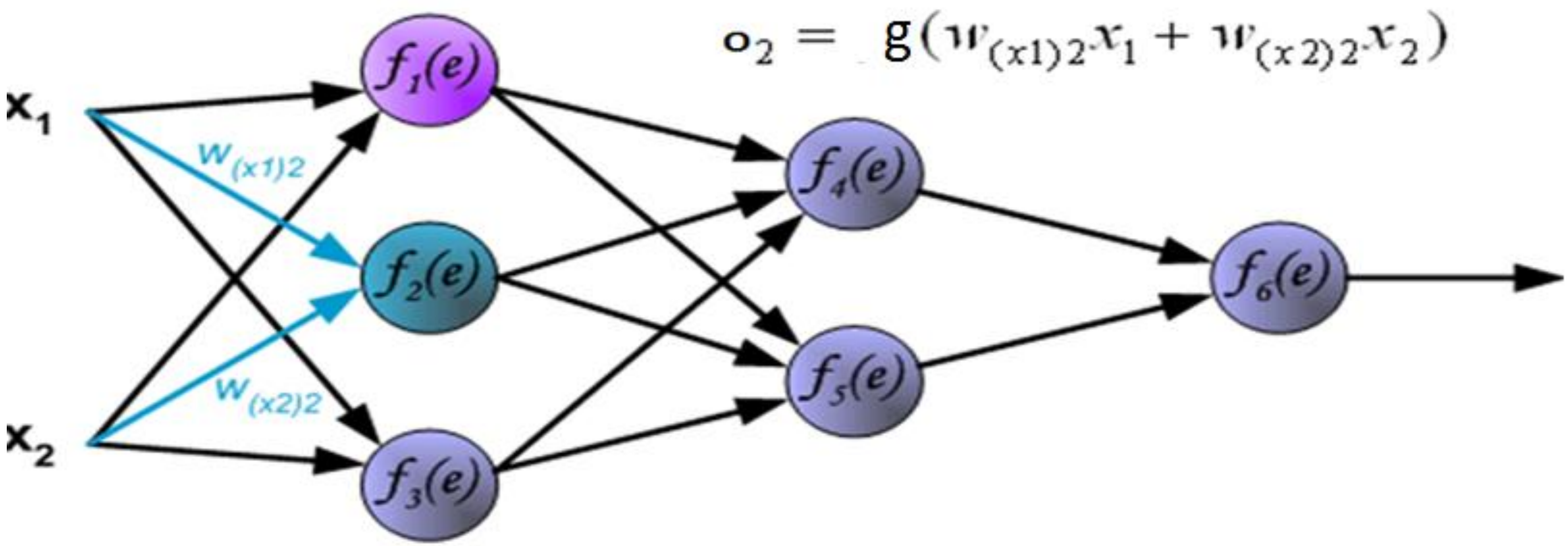
# Back propagation: visual demonstration

# 3-2-1 Neural network



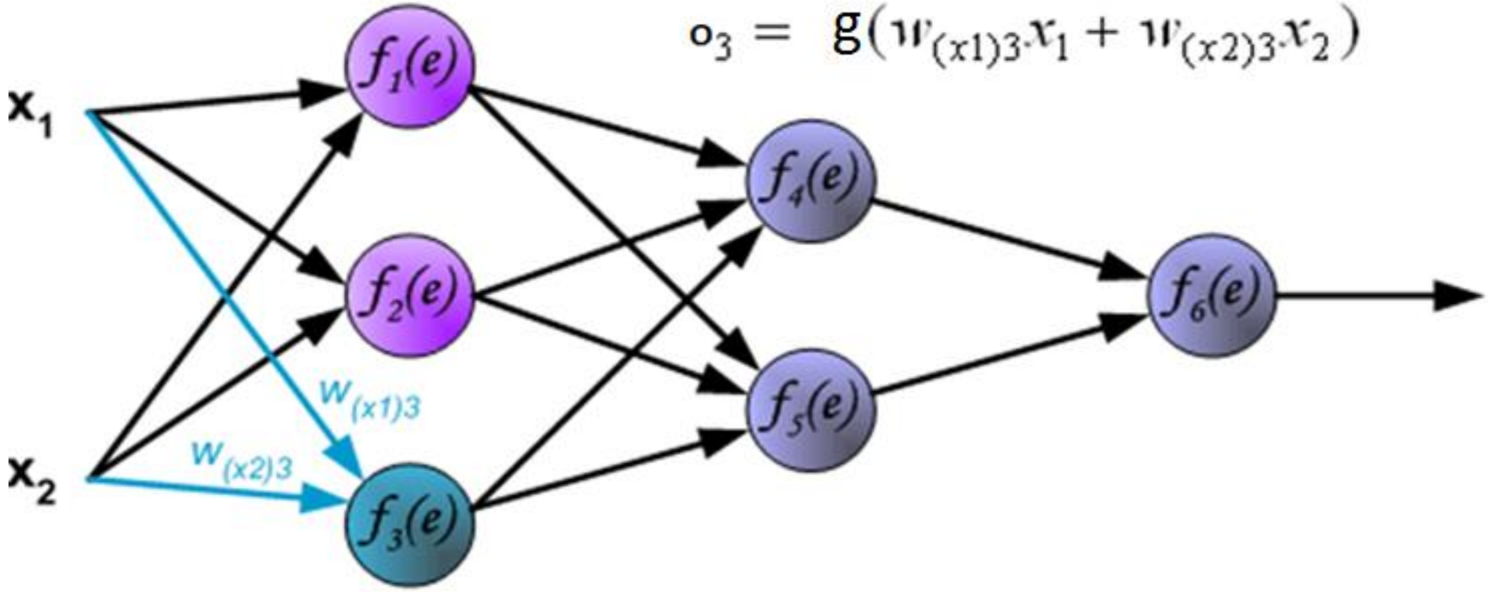
# Forward pass: calculate outputs $o_j$



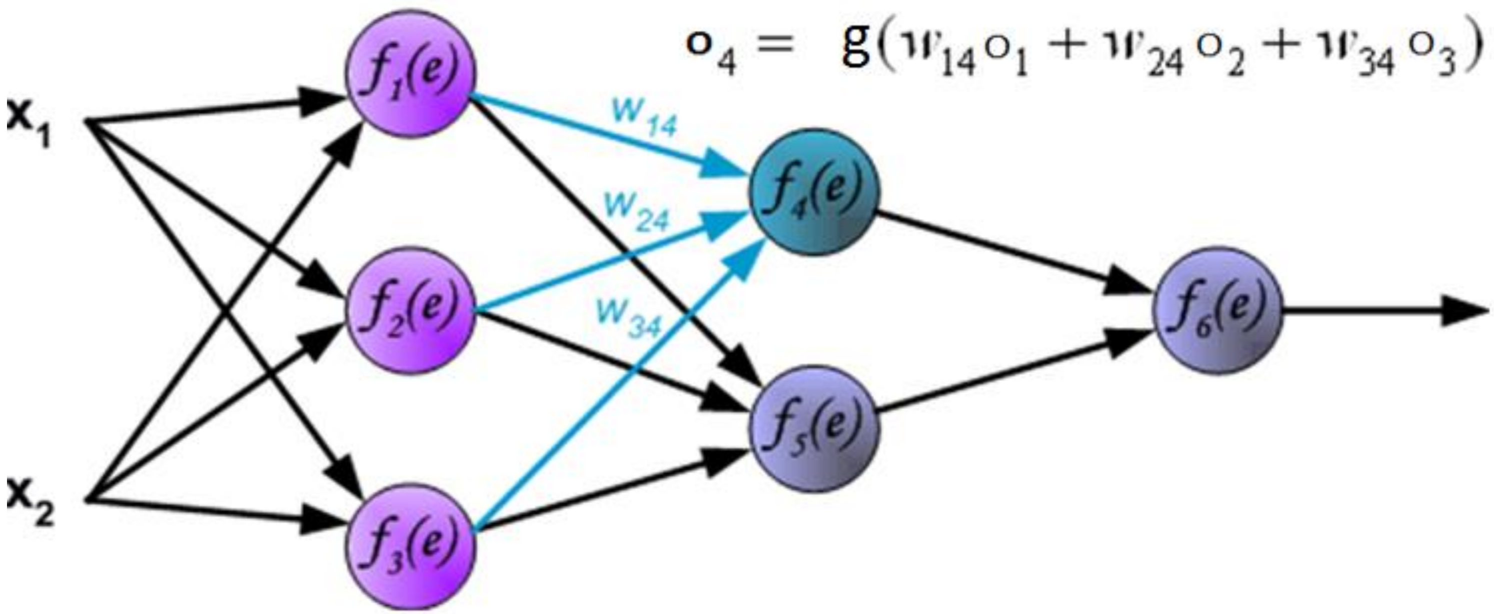



# Forward pass: calculate outputs $o_j$

$$o_3 = g(w_{(x1)3} \cdot x_1 + w_{(x2)3} \cdot x_2)$$

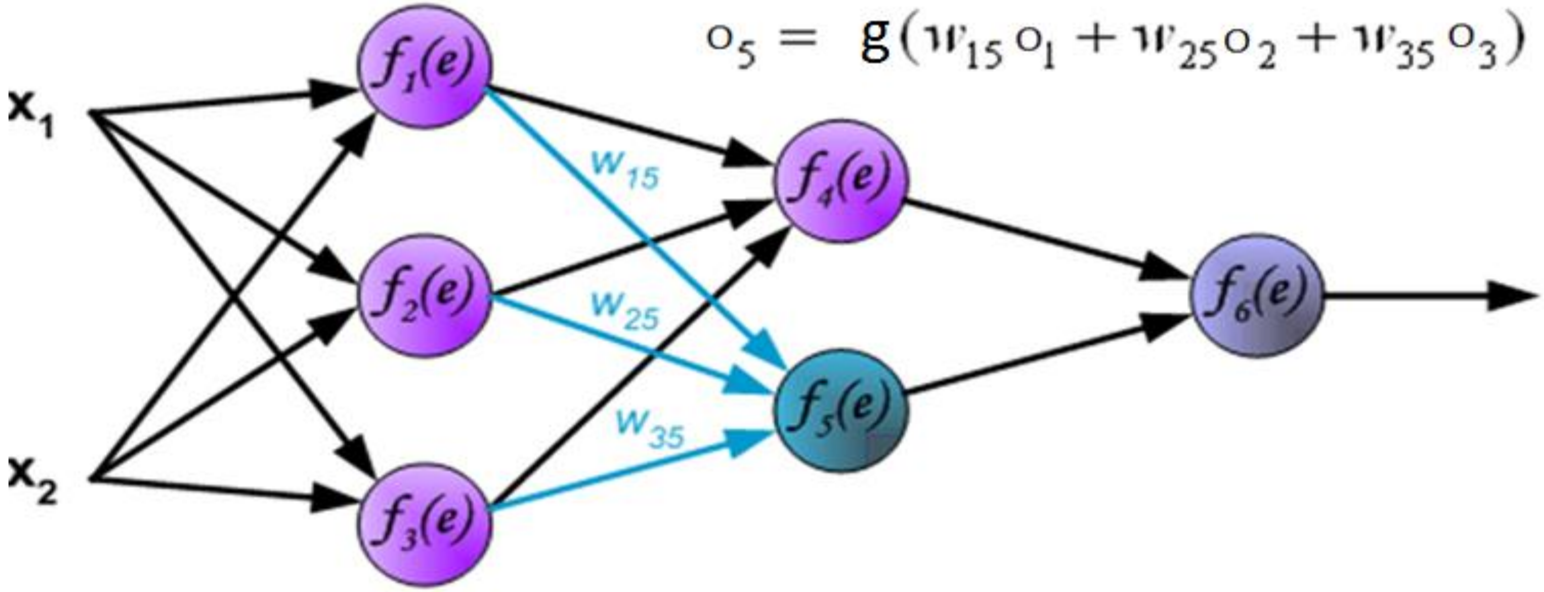



$$o_4 = g(w_{14} o_1 + w_{24} o_2 + w_{34} o_3)$$

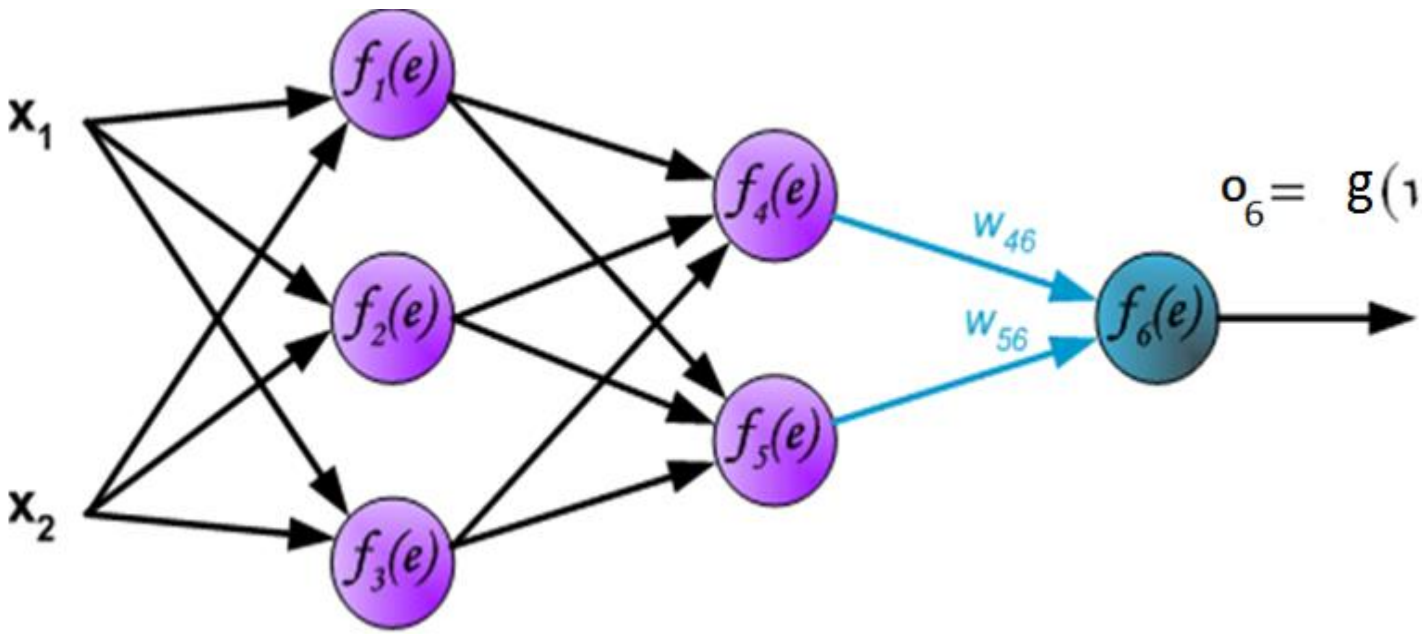


# Forward pass: calculate outputs $o_j$

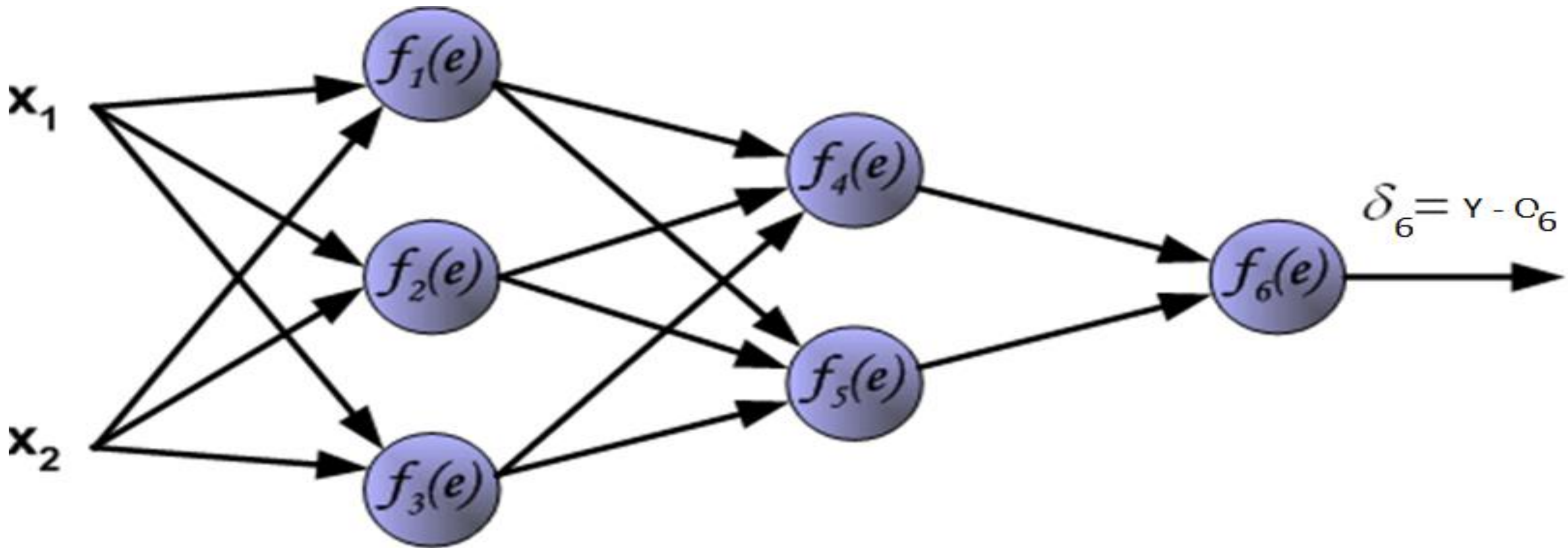
$$o_5 = g(w_{15}o_1 + w_{25}o_2 + w_{35}o_3)$$

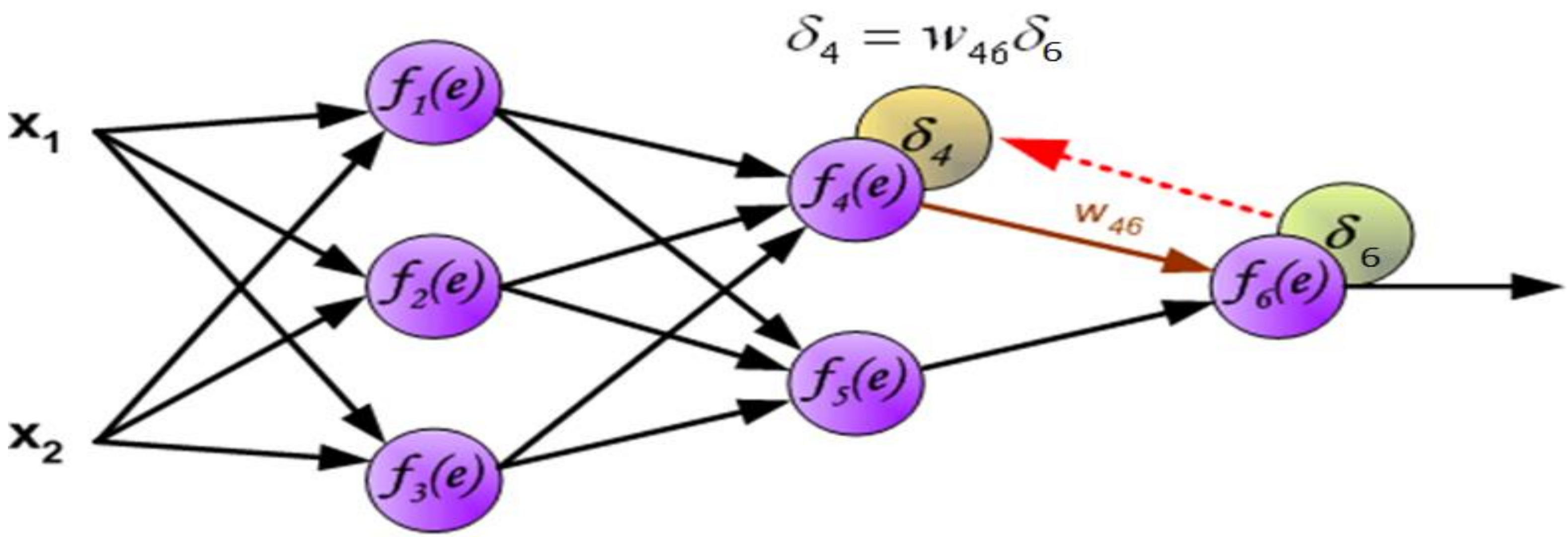



$$o_6 = g(w_{46}o_4 + w_{56}o_5)$$



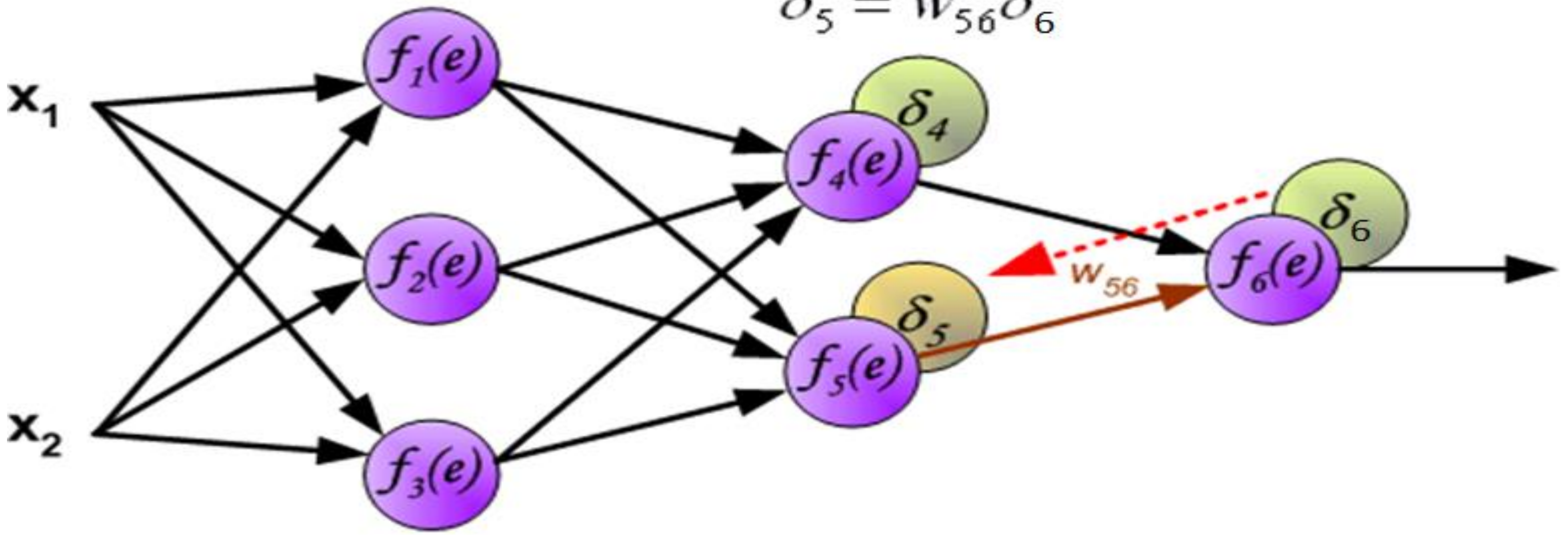
# Backward pass: calculate error grads



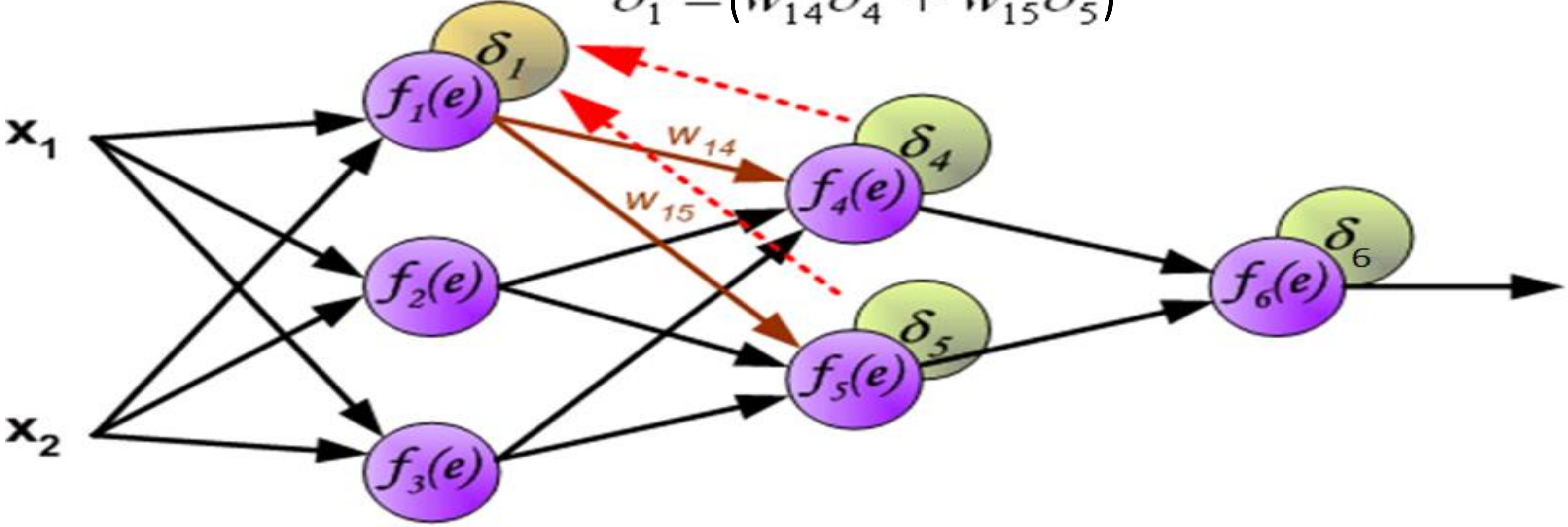



# Backward pass: calculate error grads

$$\delta_5 = w_{56}\delta_6$$

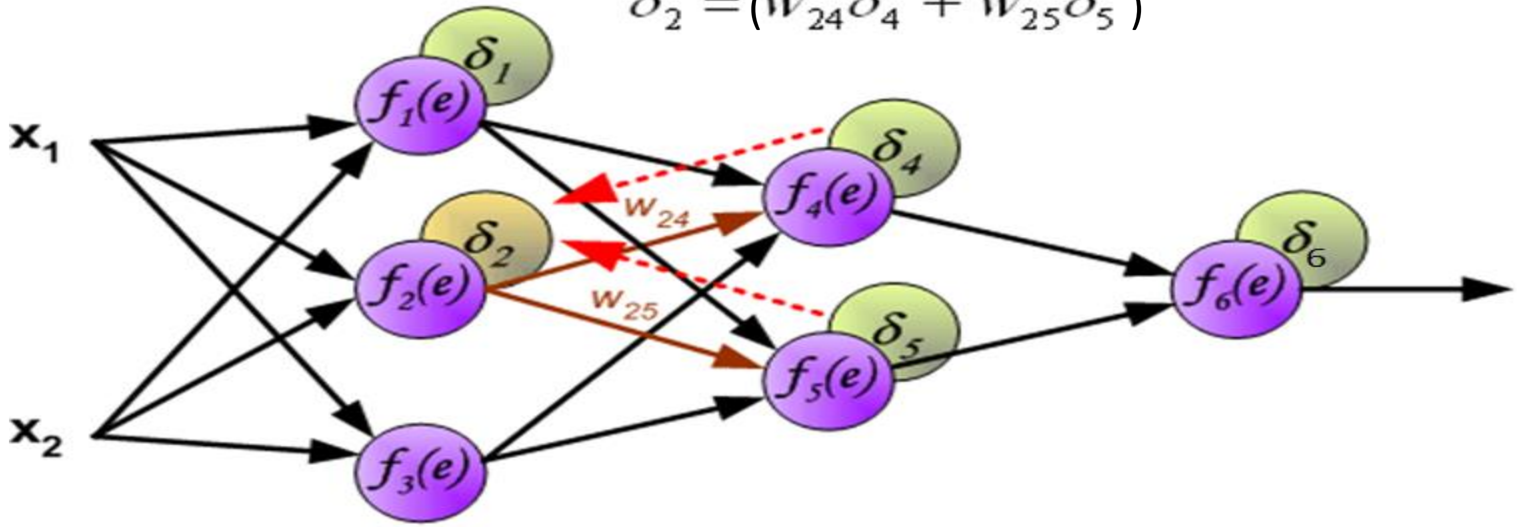



$$\delta_1 = (w_{14}\delta_4 + w_{15}\delta_5)$$

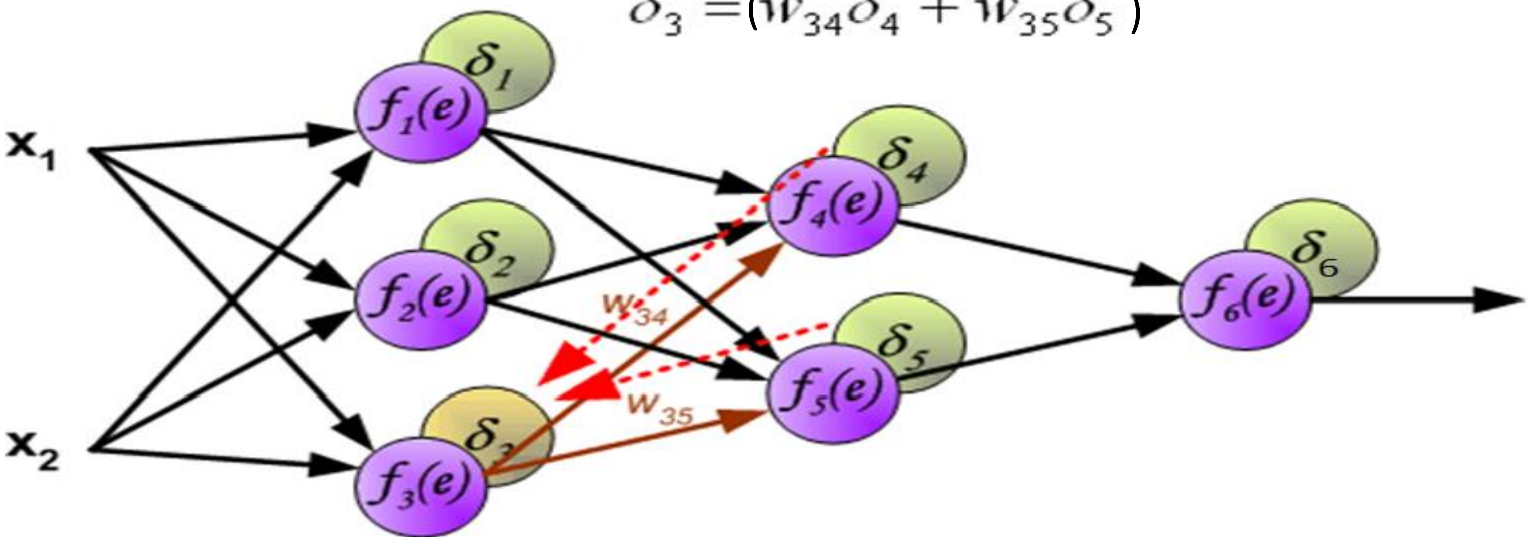


# Backward pass: calculate error grads

$$\delta_2 = (w_{24}\delta_4 + w_{25}\delta_5)$$



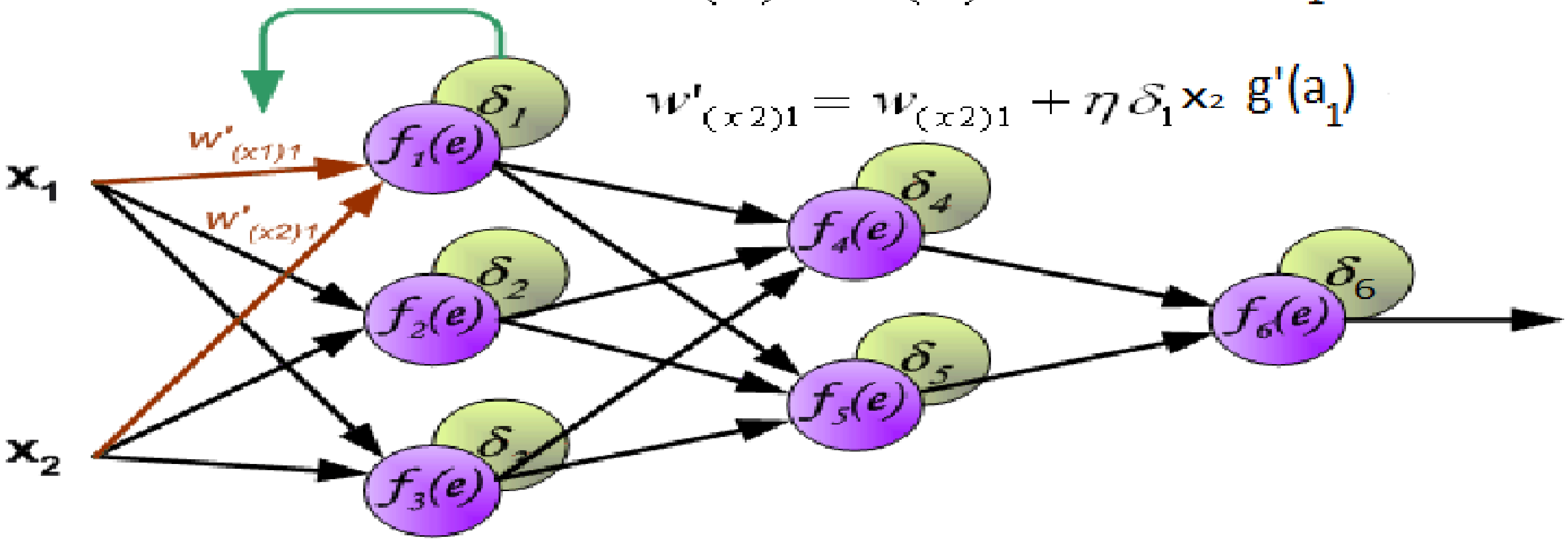

$$\delta_3 = (w_{34}\delta_4 + w_{35}\delta_5)$$



# Weight adjustment

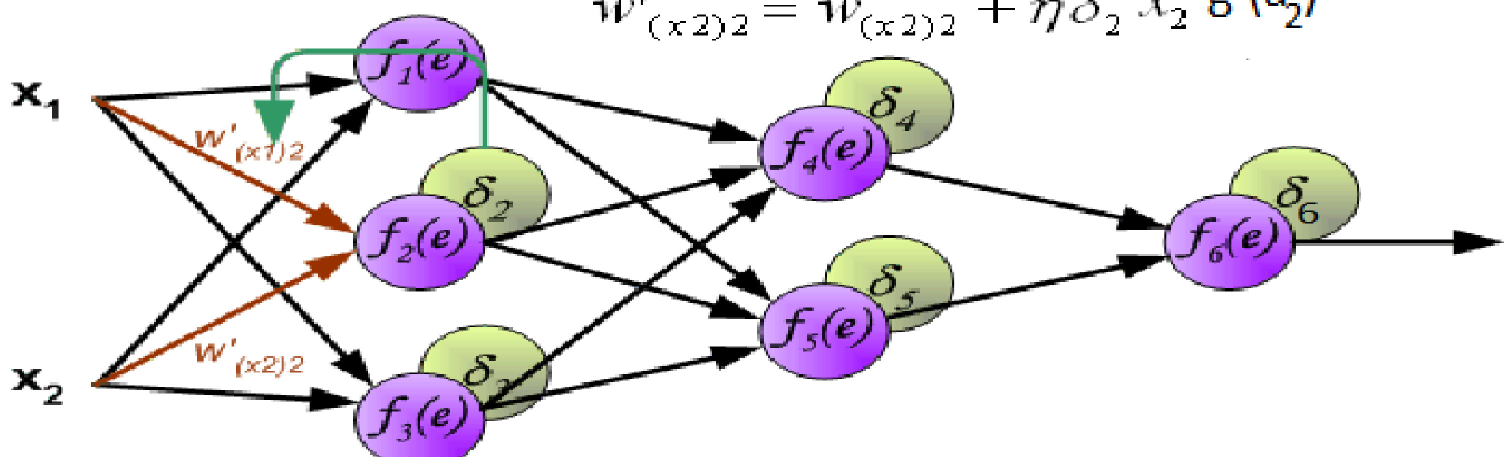
$$w'_{(x1)1} = w_{(x1)1} + \eta \delta_1 x_1 \, g'(a_1)$$

$$w'_{(x2)1} = w_{(x2)1} + \eta \delta_1 x_2 \, g'(a_1)$$




$$w'_{(x1)2} = w_{(x1)2} + \eta \delta_2 x_1 \, g'(a_2)$$

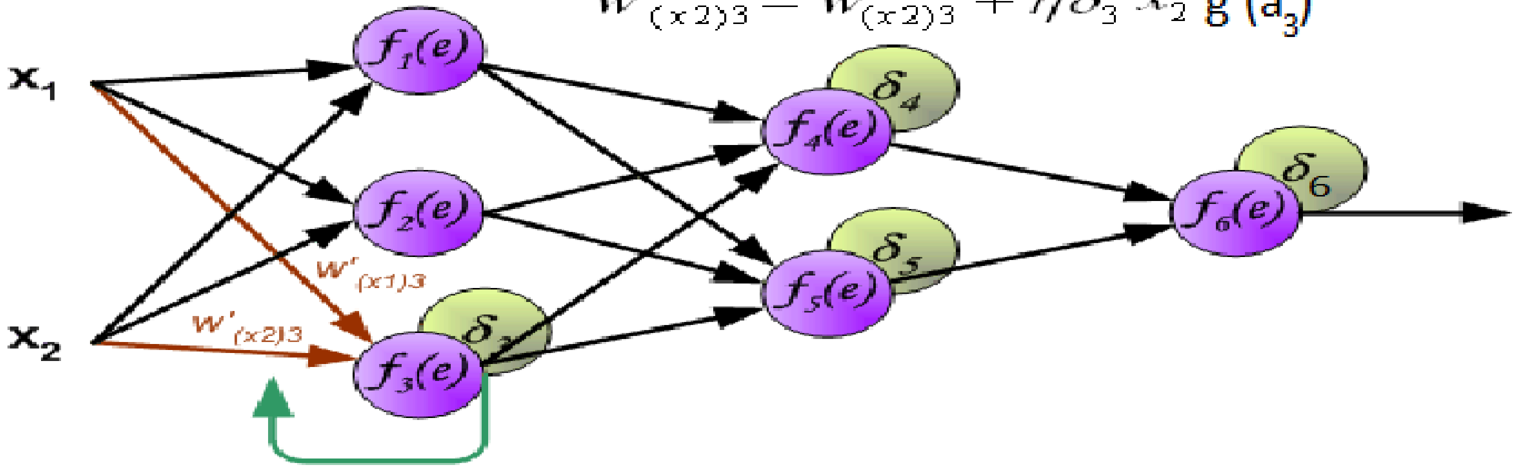
$$w'_{(x2)2} = w_{(x2)2} + \eta \delta_2 x_2 \, g'(a_2)$$



# Weight adjustment

$$w'_{(x1)3} = w_{(x1)3} + \eta\delta_3 \; x_1 \; g'(a_3)$$

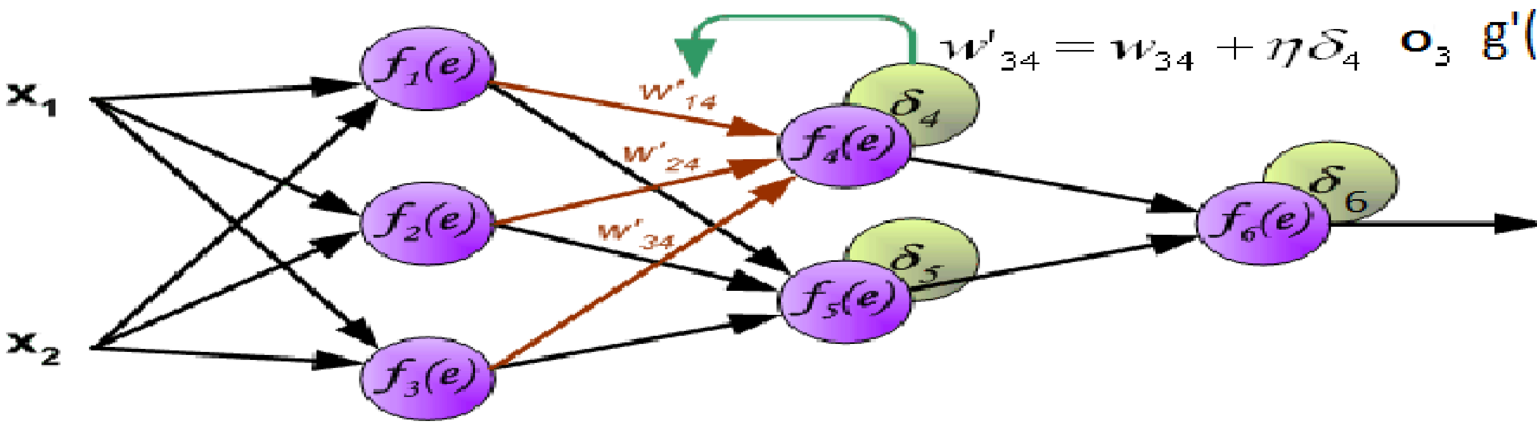
$$w'_{(x2)3} = w_{(x2)3} + \eta\delta_3 \; x_2 \; g'(a_3)$$




$$w'_{14} = w_{14} + \eta\delta_4 \; \mathbf{o}_1 \; g'(a_4)$$

$$w'_{24} = w_{24} + \eta\delta_4 \; \mathbf{o}_2 \; g'(a_4)$$

$$w'_{34} = w_{34} + \eta\delta_4 \; \mathbf{o}_3 \; g'(a_4)$$

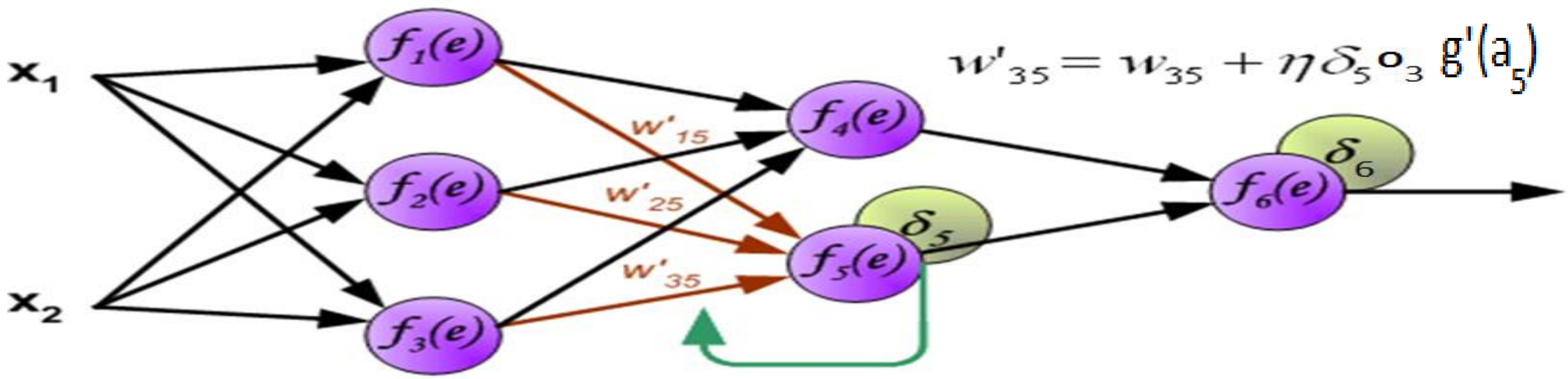


# Weight adjustment

$$w'_{15} = w_{15} + \eta \delta_5 \mathbf{o}_1 \; g'(a_5)$$

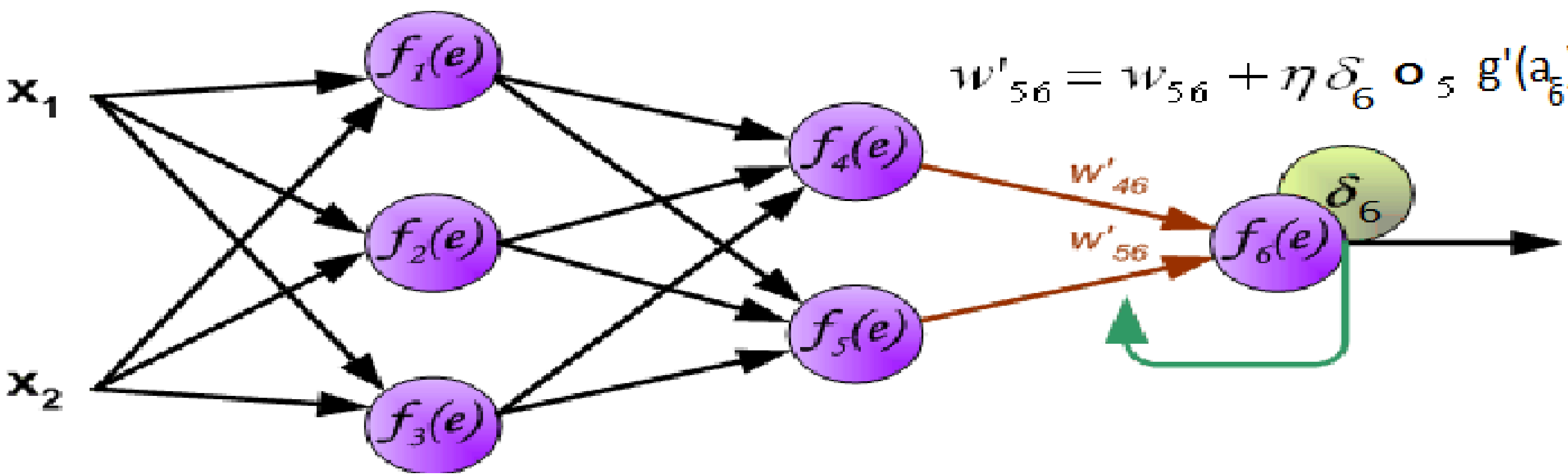
$$w'_{25} = w_{25} + \eta \delta_5 \mathbf{o}_2 \; g'(a_5)$$

$$w'_{35} = w_{35} + \eta \delta_5 \mathbf{o}_3 \; g'(a_5)$$




$$w'_{46} = w_{46} + \eta \delta_6 \; \mathbf{o}_4 \; g'(a_6)$$

$$w'_{56} = w_{56} + \eta \delta_6 \; \mathbf{o}_5 \; g'(a_6)$$



# How many train samples to be used for gradient computation?

- BGD: gradient is calculated using all the train samples
- SGD: gradient is calculated using one random train sample
- SBGD: gradient is calculated using small batch of random train samples

# NeuralNet Learning Algorithm(BGD)

Assume some random weights and random bias for each neuron in the network

Do the following until the error is below threshold or maximum number of iterations reached:

**a. Forward Pass**: Propagate the input forward through the network

Calculate the output $\boldsymbol{o}_j$ of every unit j in the network

**b. Backward Pass:** Propagate the errors backward through the network

For each network output unit *j*, calculate its error term $\delta_j$

$$\delta_j = E'(o_j)\, g'(a_j)$$

*[Sum it across all samples]*

For each hidden unit j, calculate its error term $\delta_j$

$$\delta_j = g'(a_j) \sum_k \delta_k w_{jk}$$

*[Sum it across all samples]*

**c. Weight & Bias updates:** Update weights using gradient descent rule

Update each network weight $w_{ij}$ : $w_{ij} = w_{ij} - \eta \delta_j o_i$

Note: keep $o_i = 1$ for i = 0 (bias connection)

# Issues in neural network

- Slower learning
- Local minima
- Overfitting

# Reducing the learning time: SGD version

Assume some random weights and random bias for each neuron in the network

Do the following until the error is below threshold or maximum number of iterations reached:

Choose a random training sample i = $\langle \mathbf{x}_i, \mathbf{y}_i \rangle$

**a. Forward Pass**: Propagate the input forward through the network

Calculate the output $O_j$ of every unit j in the network

**b. Backward Pass:** Propagate the errors backward through the network

For each network output unit $j$, calculate its error term $\delta_j$

$$\delta_j = E'(o_j)$$

For each hidden unit j, calculate its error term $\delta_j$

$$\delta_j = \sum_k \delta_k w_{jk}$$

**c. Weight & Bias updates:** Update weights using gradient descent rule

Update each network weight $w_{ij}$ : $w_{ij} = w_{ij} - \eta \delta_j o_i g'(a_j)$

Note: keep $o_i = 1$ for i = 0 (bias connection)

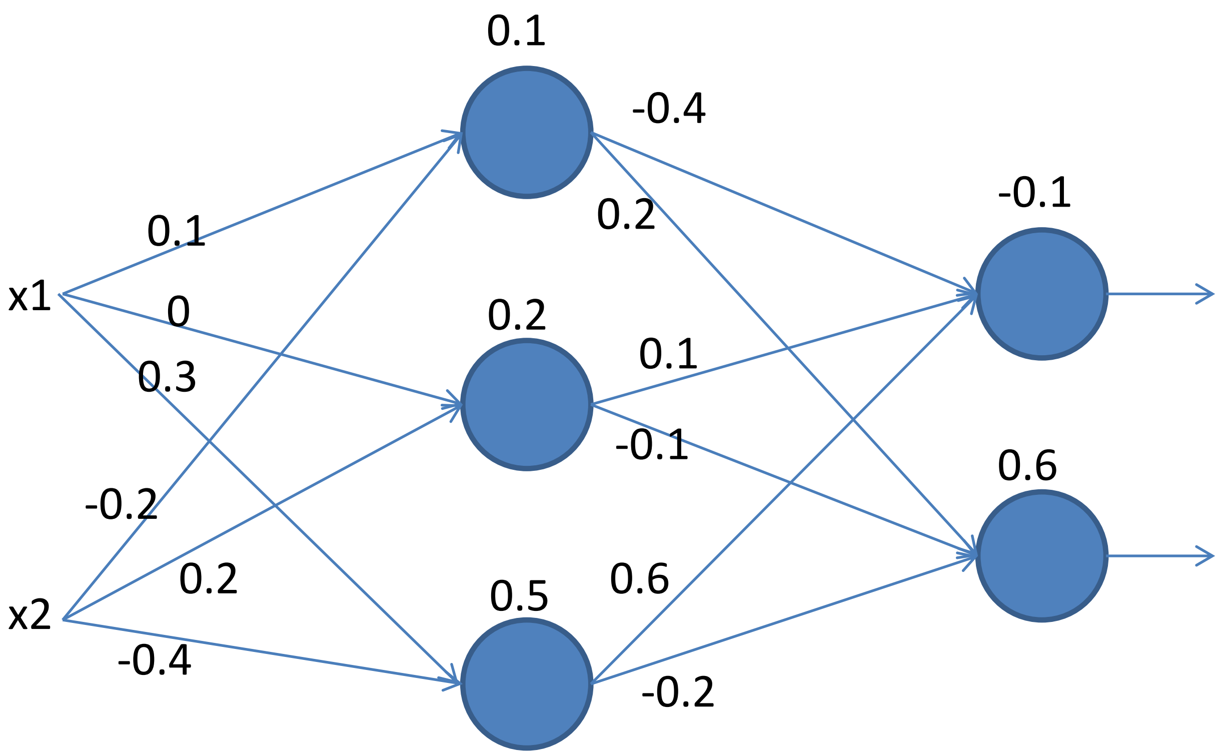
# Heuristics to avoid local minima

- Use a fixed learning rate
- Adapt the learning rate
- Add momentum

# Handling overfitting

- Use a fixed learning rate
- Adapt the learning rate
- Add momentum

# Numerical Example: Classification




**Assumptions**

- Sigmoid units
- Cross Entropy loss
- $\eta$=0.1
- Initial weights and biases as given in fig
- Output Encoding
  - class1:10
  - class2:01

| x1 | x2 | Output |
|---|---|---|
| 0.6 | 0.1 | Class1 |
| 0.2 | 0.3 | Class2 |
| | | |