CS 412

FEB 27TH – BACKPROPAGATION

Perceptron

What is the problem with the simple perceptron?

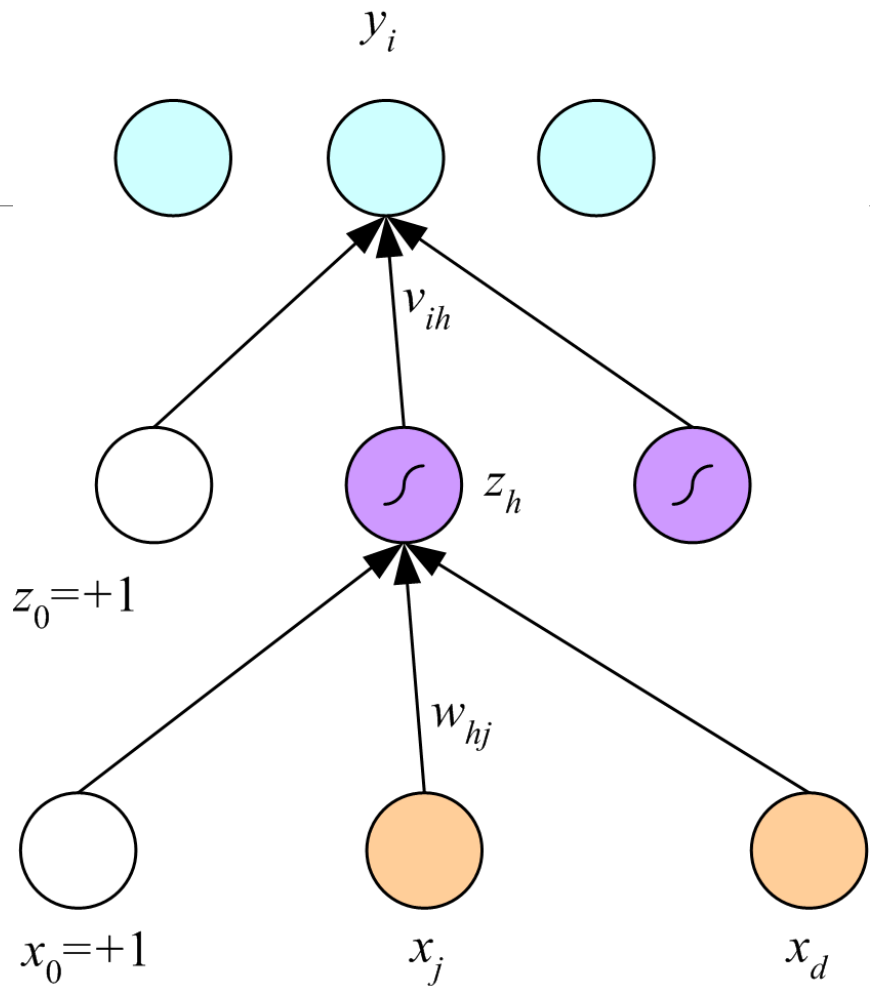
- It can't model non-linear data

How do we fix this?

- SVM fixed this by using the kernel methods
- Can the perceptron? **Yes**, but that's not what the neural network does

Let's add multiple layers to the perceptron

- At each level we have a **regression** model defined by the activation function and **always** a constant w_0



How to train?

$j \in \mathbb{R}^d$

We (hopefully) have some idea of how a particular set of weights causes the neural network to make a decision

- We have some vector of inputs X_d that are all fed as parameters to some number of nodes
- Each of these nodes outputs a sigmoid function to the next hidden layer
- This process eventually leads to the final layer, which makes the final prediction

activation

for binary problems - step

- ReLU

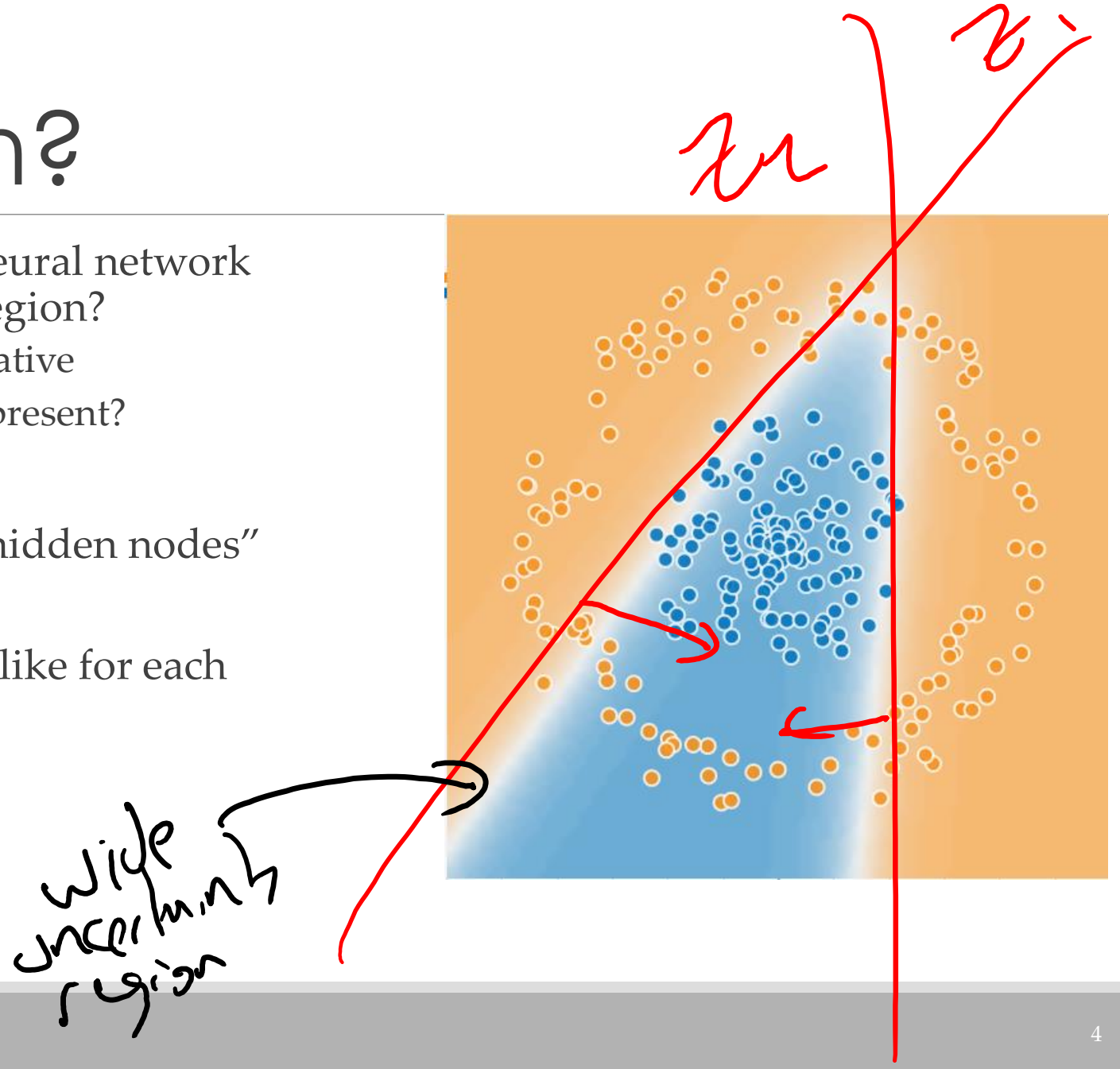
How to train?

What is the structure of the neural network that produced this decision region?

- Blue is positive, orange is negative
- What do the white regions represent?

Two lines from two middle "hidden nodes" with sigmoid behavior

What might the weights look like for each of these nodes?



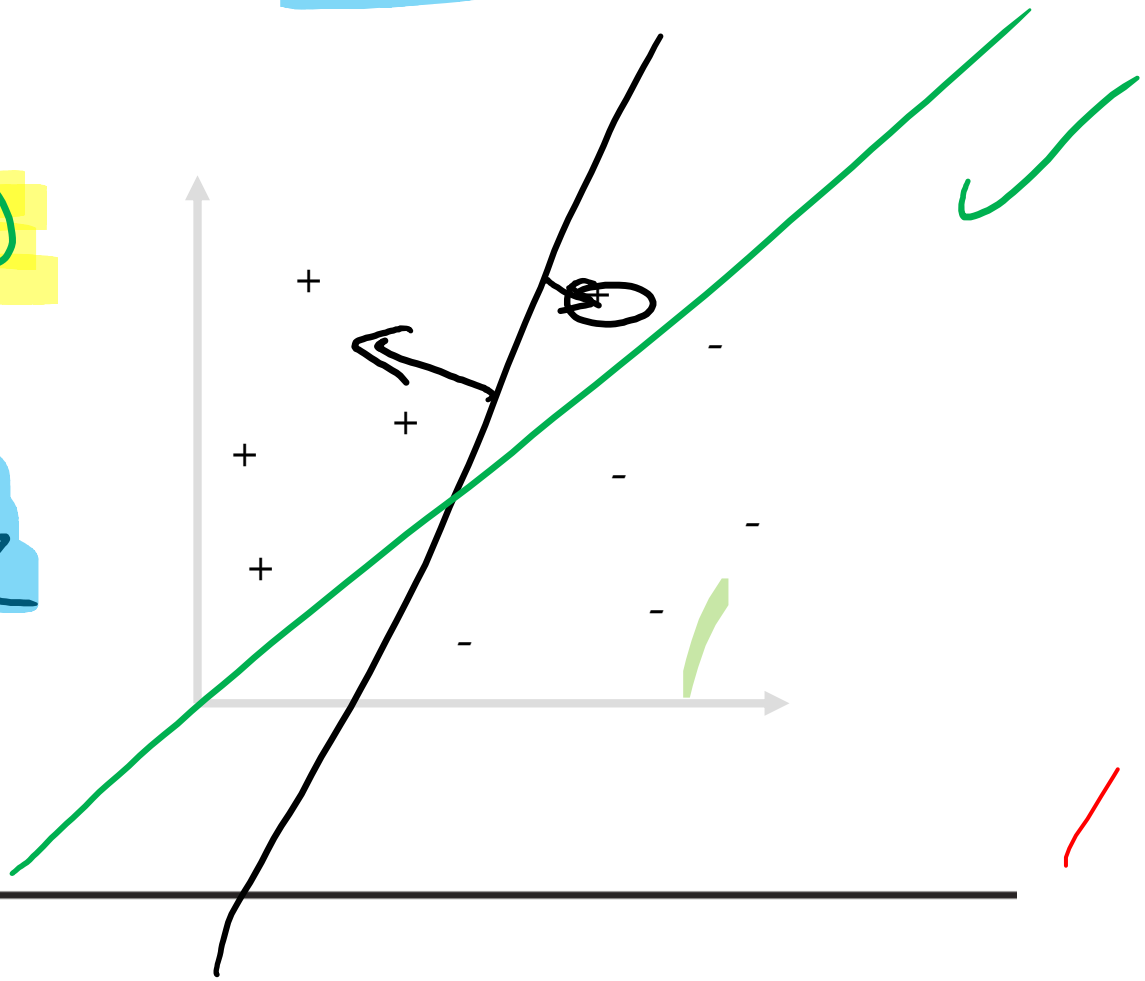
Algorithm 8.4: Perceptron algorithm

Input: linearly separable data set $\mathbf{x}_i \in \mathbb{R}^D$, $y_i \in \{-1, +1\}$ for $i = 1 : N$;
Initialize $\boldsymbol{\theta}_0$;
$k \leftarrow 0$;
repeat
    $k \leftarrow k + 1$;
    $i \leftarrow k \bmod N$;
    if $\hat{y}_i \neq y_i$ **then**
        $\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k + y_i \mathbf{x}_i$
    else
        no-op
until *converged*;

estimated

$\eta = 1/2$

$(\hat{y} - y)$



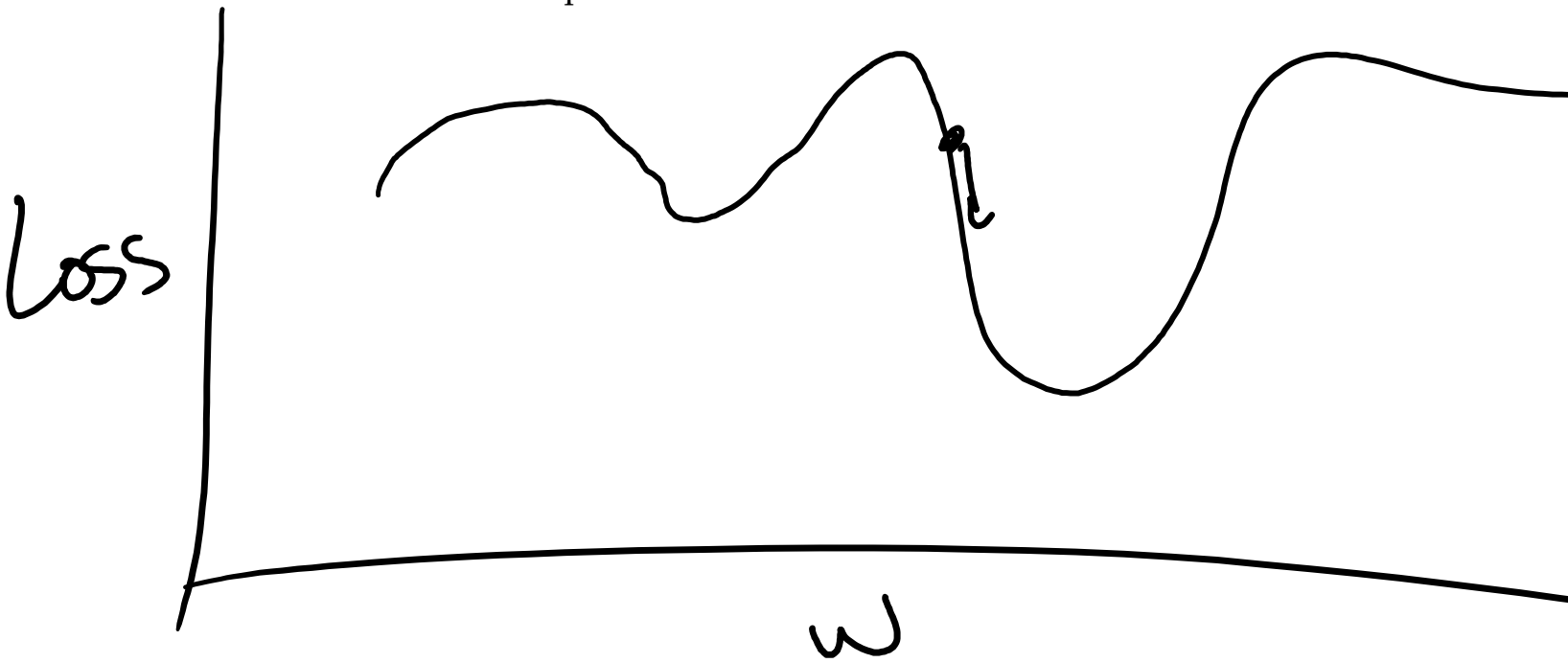
What's wrong with this approach?

It's an easy way to separate linear data. But what's wrong?

- Data isn't always linearly separable
- Nodes of the neural network work in conjunction with each other
  - This approach would mean that all internal nodes convert to the same point!

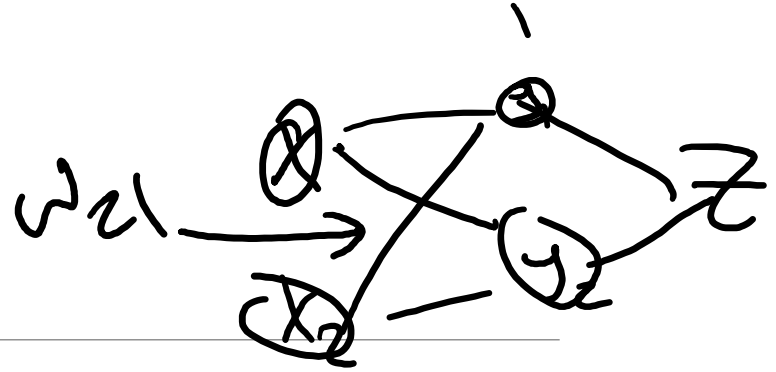
How do we get around this?

- Random weights
- Feedback between nodes



WTF $\frac{\partial \ell(W)}{\partial w_k}$

W



How to Train

- Training to minimize sum-squared error

superscript

$$\ell(W) = \frac{1}{2}\sum_j [y^j - g(w_0 + \sum_i w_i x_i^j)]^2$$

1 layer simple perceptron

j = datapoints

i = features

x_i^j = i^{th} feature j^{th} point

g = activation function

How to Train

- Training to minimize sum-squared error

$$\ell(W) = \frac{1}{2}\sum_j [y^j - g(w_0 + \sum_i w_i x_i^j)]^2$$

$$\frac{\partial \ell(W)}{\partial w_k} = -\sum_j [y^j - g(w_0 + \sum_i w_i x_i^j)]\ x_k^j\ g'(w_0 + \sum_i w_i x_i^j)$$

chain rule

the easier it is to take derivation for

Δw_k

j ?

How to Train

out(x)

- Training to minimize sum-squared error

$$\ell(W) = \frac{1}{2}\sum_j [y^j - g(w_0 + \sum_i w_i x_i^j)]^2$$

$$\frac{\partial \ell(W)}{\partial w_k} \propto \Delta w_k \leftarrow \eta$$

Gradient descent for 1-hidden layer – Back-propagation: Computing $\frac{\partial \ell(W)}{\partial w_i^k}$

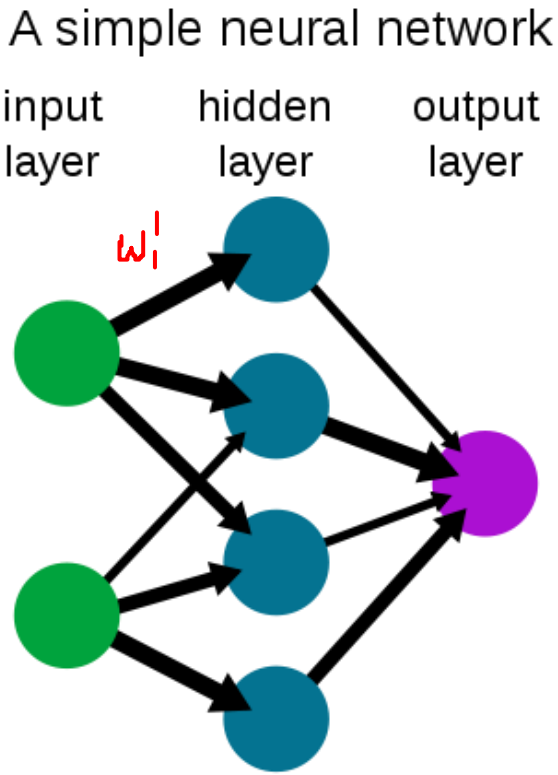
$$\ell(W) = \frac{1}{2}\sum_j [y^j - out(\mathbf{x}^j)]^2$$

Vector of length = # features

$$out(\mathbf{x}) = g\left(\sum_{k'} w_{k'} g(\sum_{i'} w_{i'}^{k'} x_{i'})\right)$$

$$\frac{\partial \ell(W)}{\partial w_i^k} = \sum_{j=1}^{m} -[y - out(\mathbf{x}^j)]\frac{\partial out(\mathbf{x}^j)}{\partial w_i^k}$$

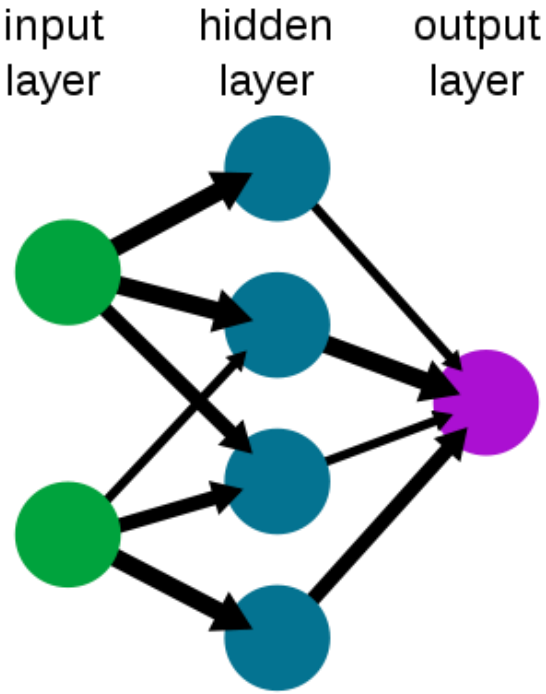
Dropped w_0 to make derivation simpler



We can do for first layer

Back-propagation Algorithm

A simple neural network



$$out(\mathbf{x}) = g\left(w_0 + \sum_k w_k g(w_0^k + \sum_i w_i^k x_i)\right)$$

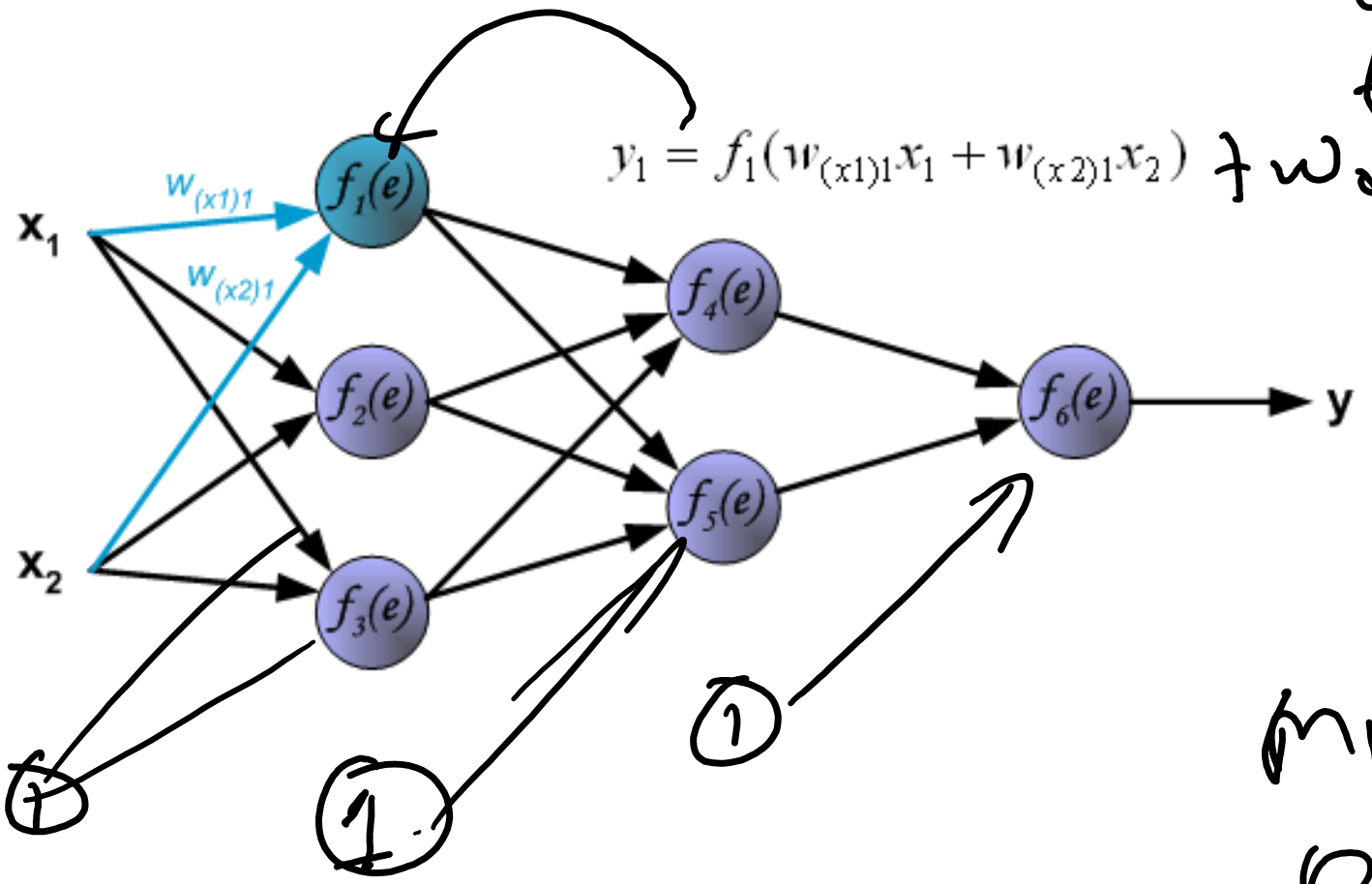
Sigmoid function: $g(z) = \frac{1}{1 + exp(-z)}$

(Rummelhart et al. 1986)

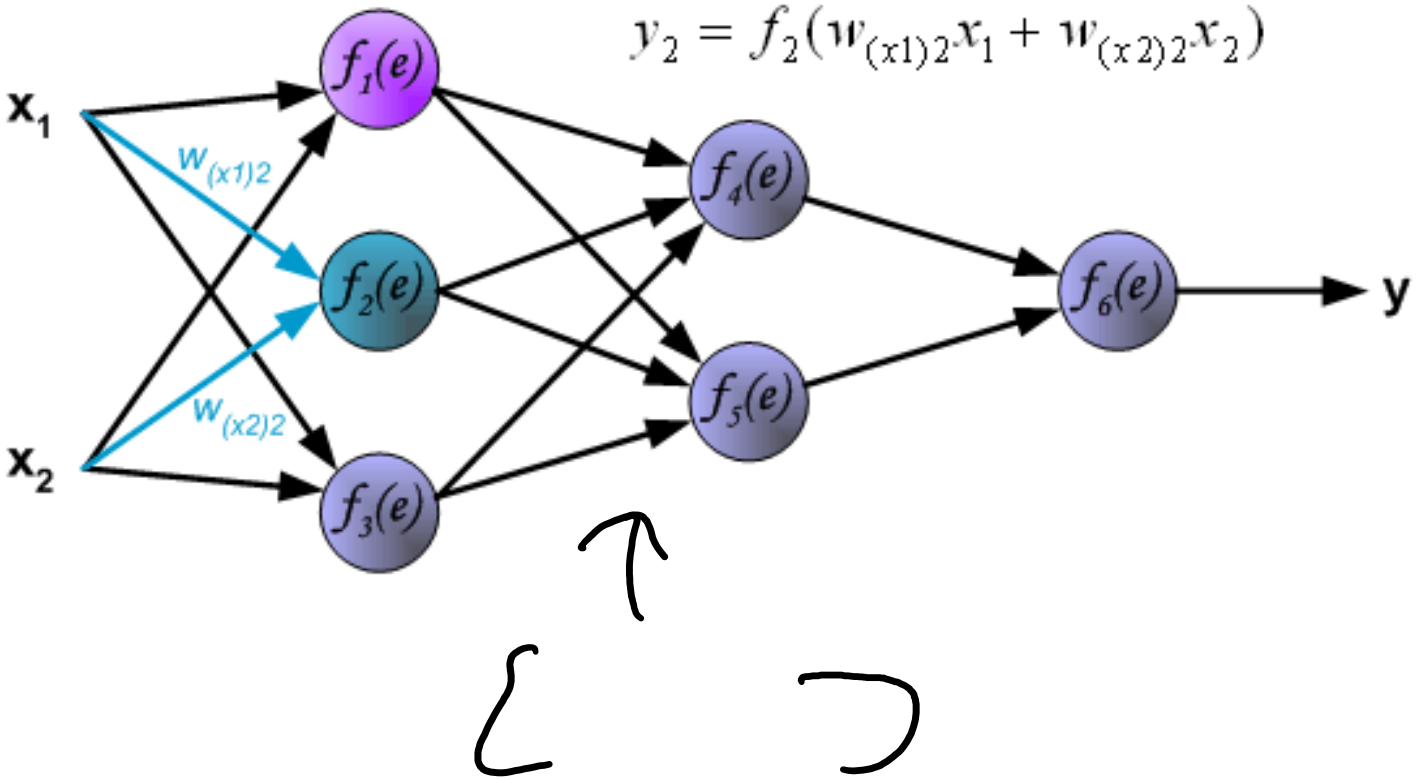
Non-convex function of '**w**'s

Back-prop: find local optima

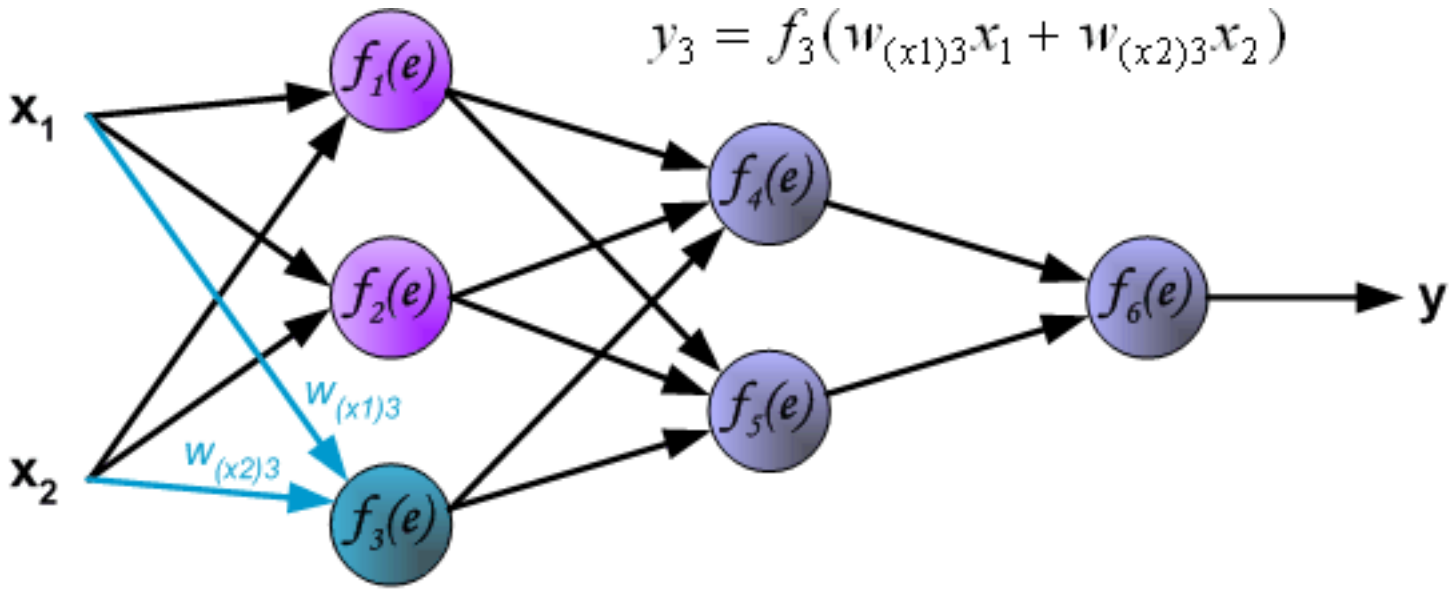
Learning Algorithm: Backpropagation



Learning Algorithm: Backpropagation

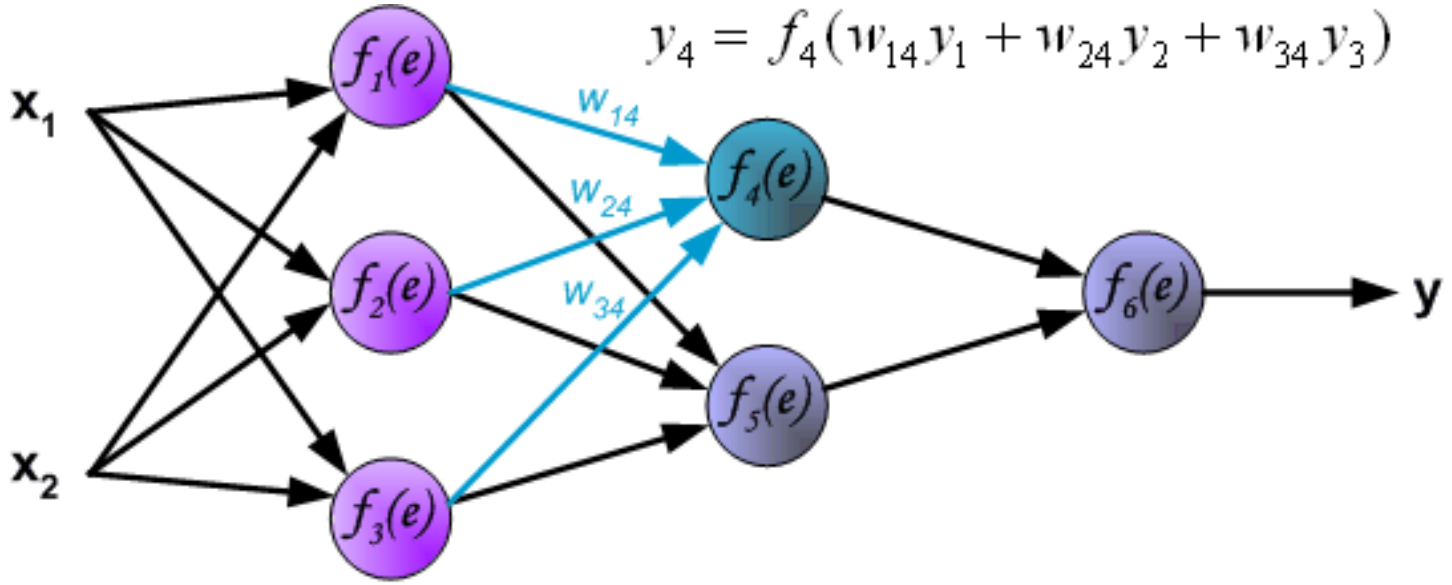


Learning Algorithm: Backpropagation

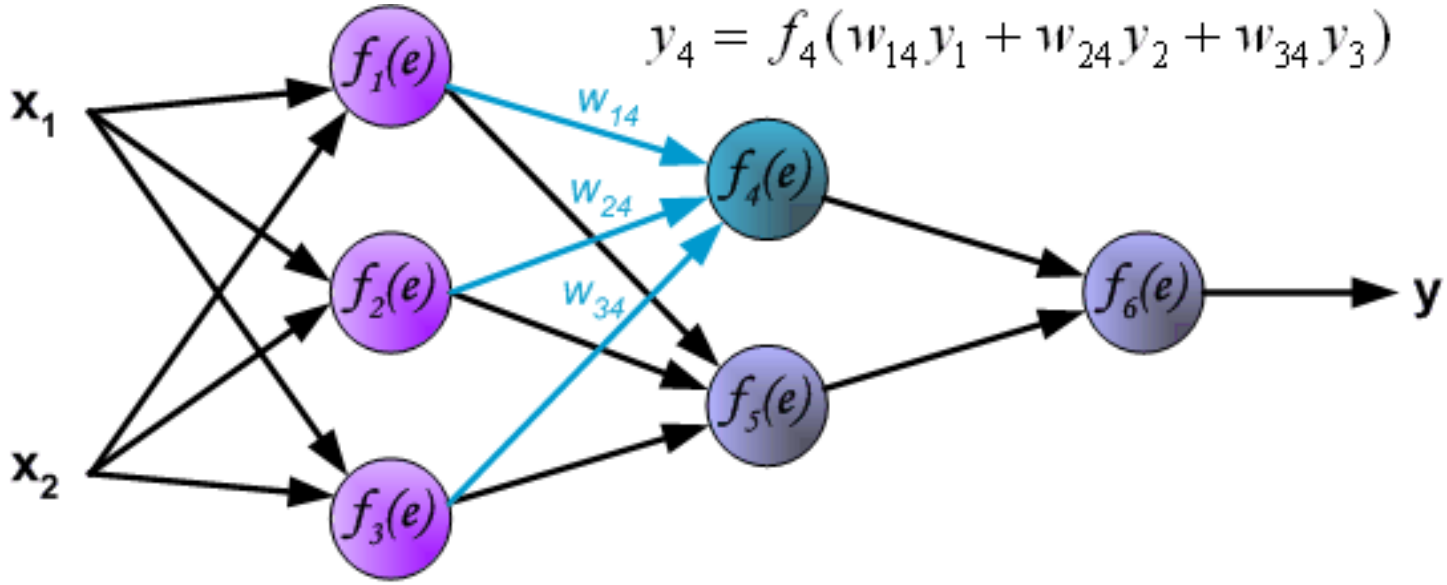


Learning Algorithm: Backpropagation

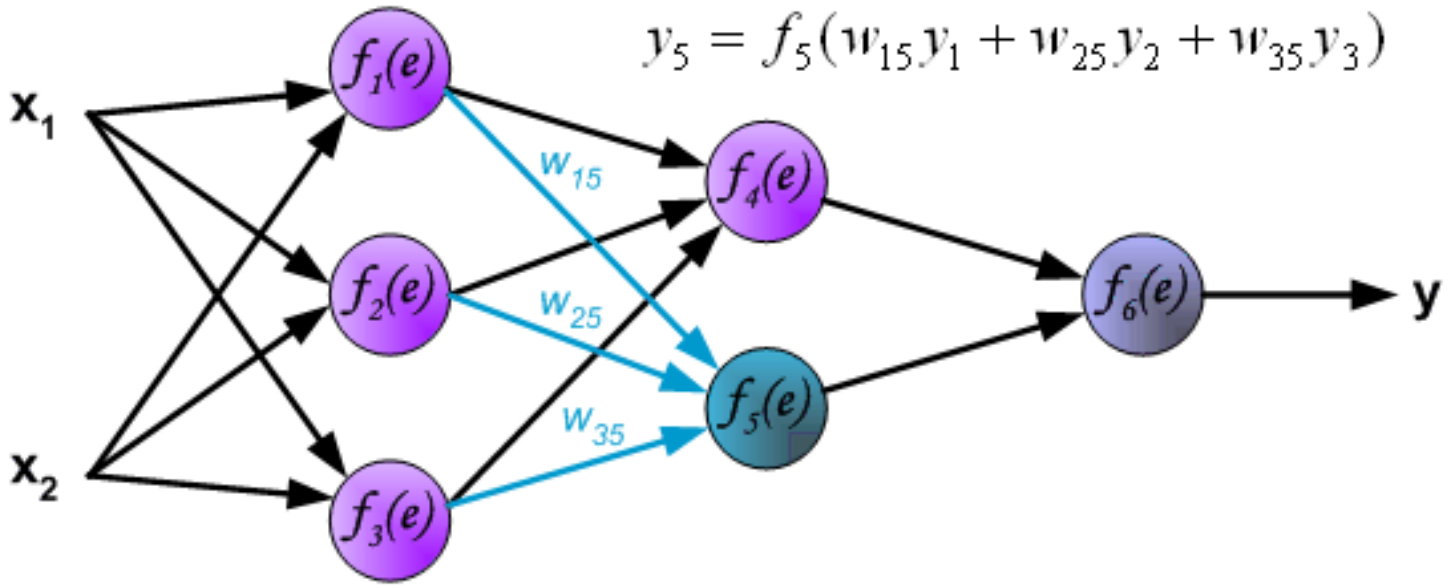
Propagation of signals through the hidden layer. Symbols w_{mn} represent weights of connections between output of neuron *m* and input of neuron *n* in the next layer.



Learning Algorithm: Backpropagation

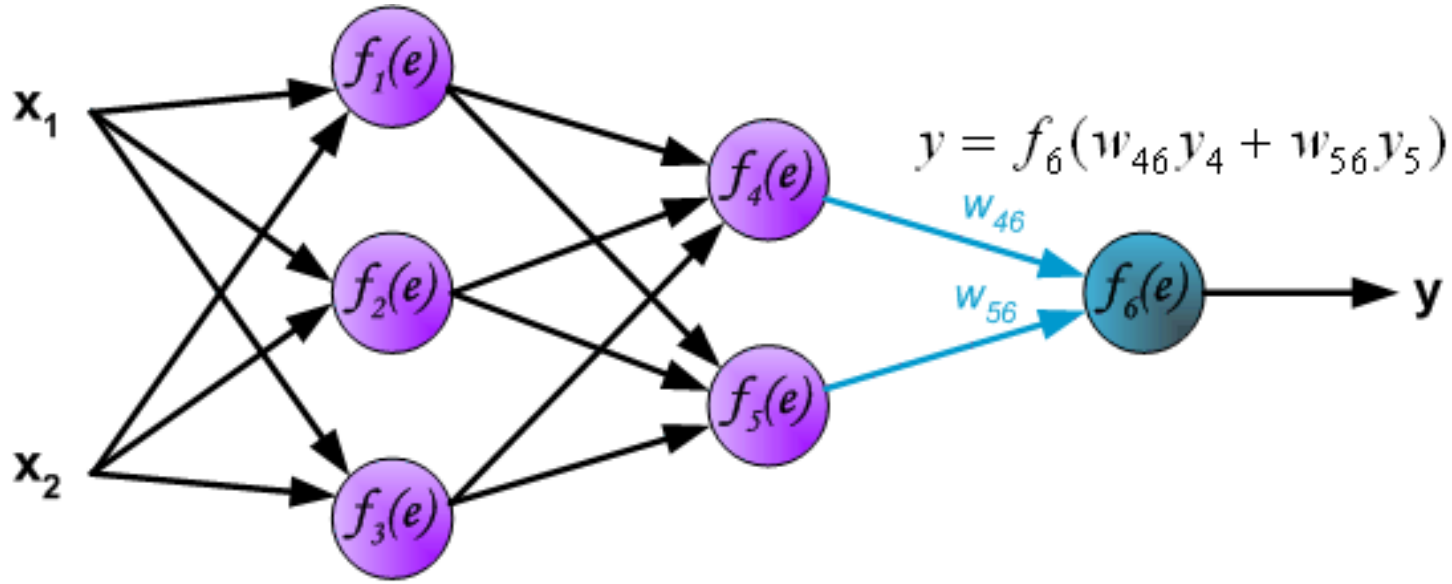


Learning Algorithm: Backpropagation



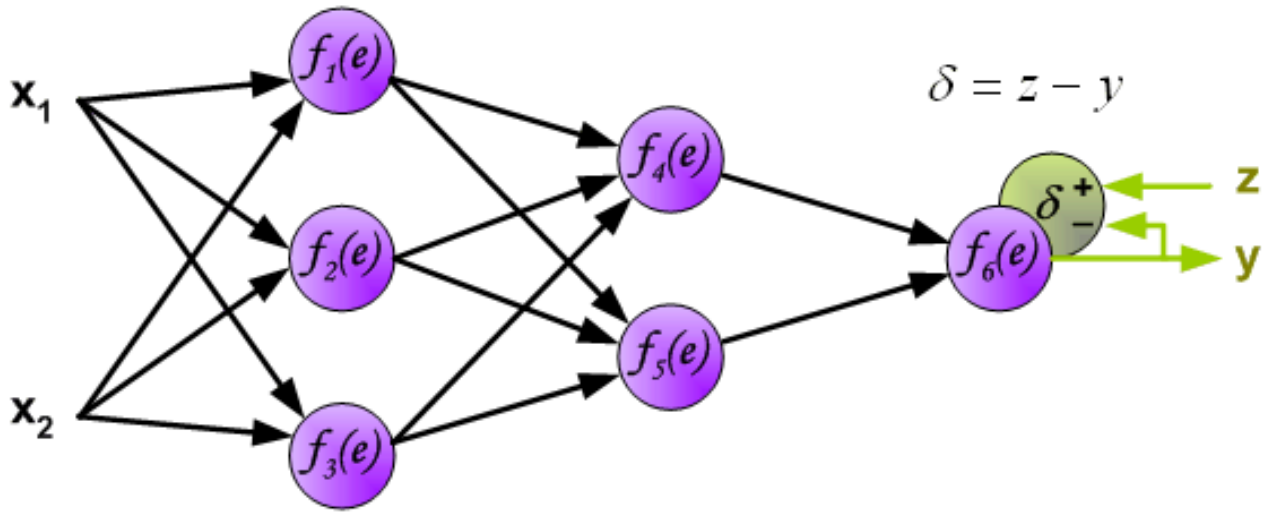
Learning Algorithm: Backpropagation

Propagation of signals through the output layer.



Learning Algorithm: Backpropagation

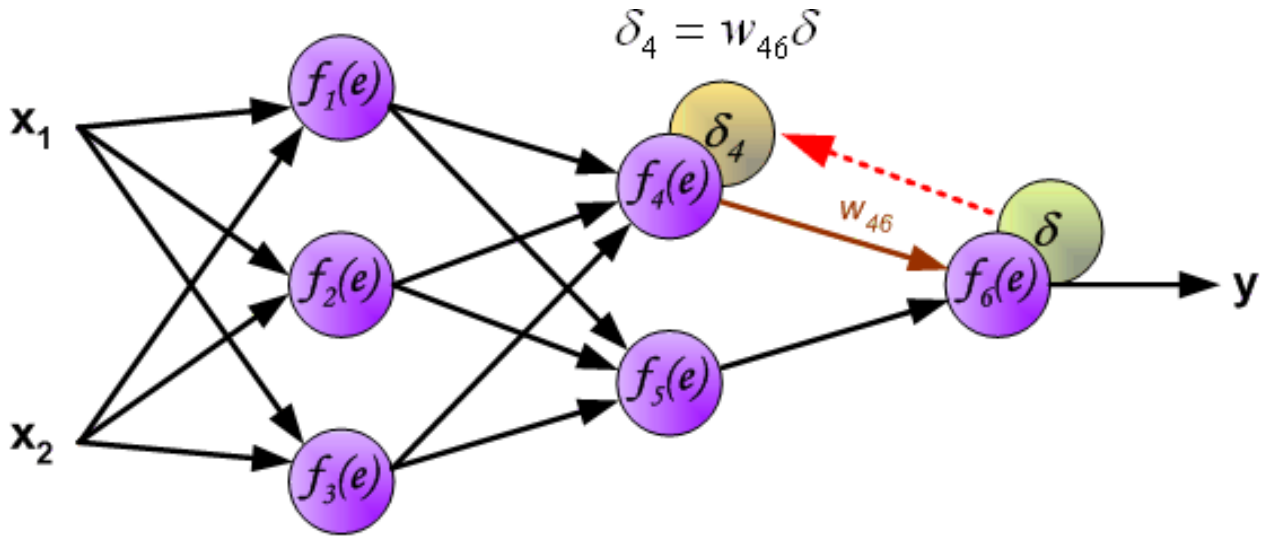
In the next algorithm step the output signal of the network *y* is compared with the desired output value (the target), which is found in training data set. The difference is called error signal *d* of output layer neuron



δ
Sensitivity

Learning Algorithm: Backpropagation

The idea is to propagate error signal *d* (computed in single teaching step) back to all neurons, which output signals were input for discussed neuron.

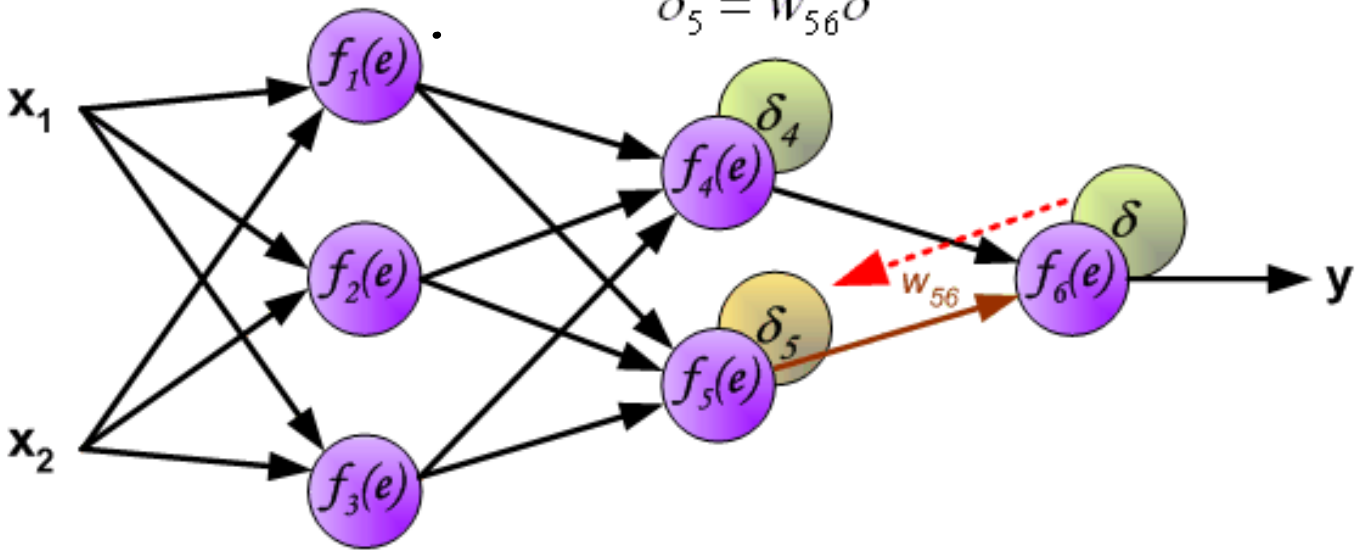


Learning Algorithm: Backpropagation

The idea is to propagate error signal *d* (computed in single teaching step) back to all neurons, which output signals were input for discussed neuron.

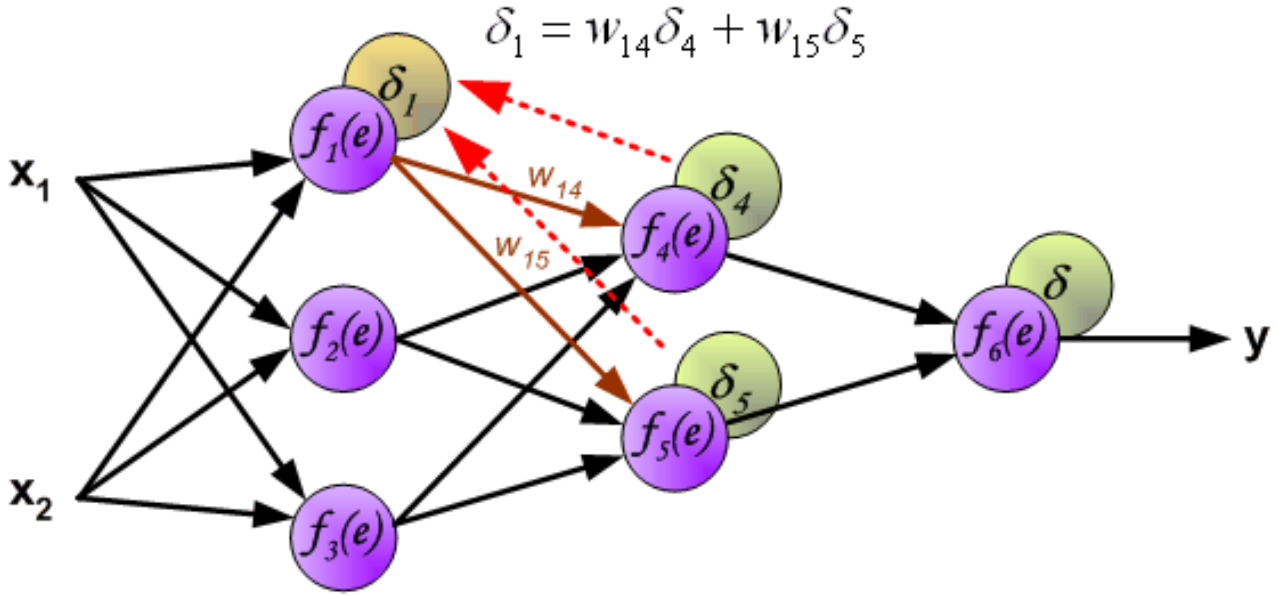
$\delta_1 = w_{14}\delta_4$

$\delta_5 = w_{56}\delta$



Learning Algorithm: Backpropagation

The weights' coefficients w_{mn} used to propagate errors back are equal to this used during computing output value. Only the direction of data flow is changed (signals are propagated from output to inputs one after the other). This technique is used for all network layers. If propagated errors came from few neurons they are added. The illustration is below:



Learning Algorithm: Backpropagation

When the error signal for each neuron is computed, the weights coefficients of each neuron input node may be modified. In formulas below *df(e)/de* represents derivative of neuron activation function (which weights are modified)

$$w'_{(x1)1} = w_{(x1)1} + \eta \delta_1 \frac{df_1(e)}{de} x_1$$

$$w'_{(x2)1} = w_{(x2)1} + \eta \delta_1 \frac{df_1(e)}{de} x_2$$