Back-propagation Algorithm: Explanation and Implementation

Shan He

School for Computational Science
University of Birmingham

Module 06-27818 and 27819:
Introduction to Neural Computation (Level 4/M)
Neural Computation (Level 3)

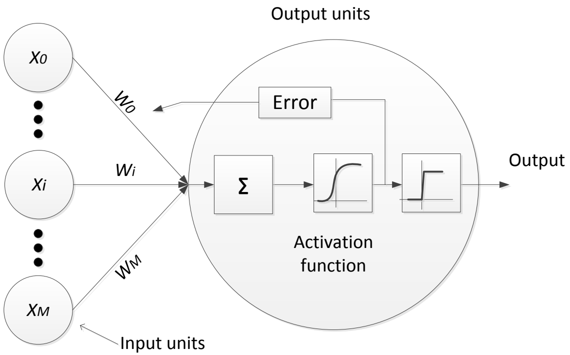
Outline of Topics

Review: the generic BP algorithm

Example: Solving classification using a simple 3-layer network

Reminder: Gradient Descent Learning

- **Purpose of NN learning**: minimise the output errors on a particular set of training data by adjusting the network weights **w**:



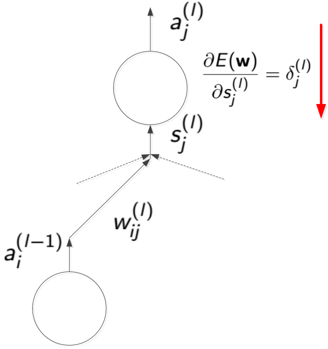
Reminder: new definition

- Let's define $s_j^{(l)} = \sum_i w_{ij}^{(l)} a_i^{(l-1)}$, which can be seen as the signal to node j which is the weighted sum of all activations of neurons of the previous layer
- For activation functions:
  - Output layer:

$$a_j^{(L)} = y = f^{(L)}\left(s_j^{(L)}\right)$$

  - Any hidden layer:

$$a_j^{(l-1)} = f^{(l-1)}\left(s_j^{(l-1)}\right)$$



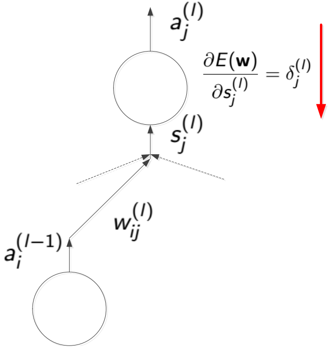
Reminder: BP algorithm

- We define:

$$\frac{\partial E(\mathbf{w})}{\partial s_j^{(l)}} = \delta_j^{(l)}$$

- Intuition: how error change with respect to the signal to the neuron j – back-propagated error signal
- The general weight updates equation for any weight at any layer

$$\Delta w_{ij}^{(l)} = \frac{\partial E(\mathbf{w})}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} a_i^{(l-1)}$$



Backpropogation algorithm: δ_j for the output/hidden layers

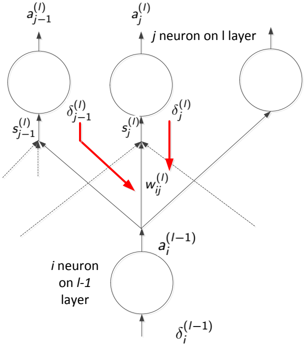
- For neuron j on the output (final) layer $l = L$:

$$\delta_j^{(L)} = \frac{\partial E(\mathbf{w})}{\partial s_j^{(L)}}$$

- For neuron i at any **hidden layer** $l-1$:

$$\delta_i^{(l-1)} = \frac{\partial E(\mathbf{w})}{\partial s_i^{(l-1)}}$$

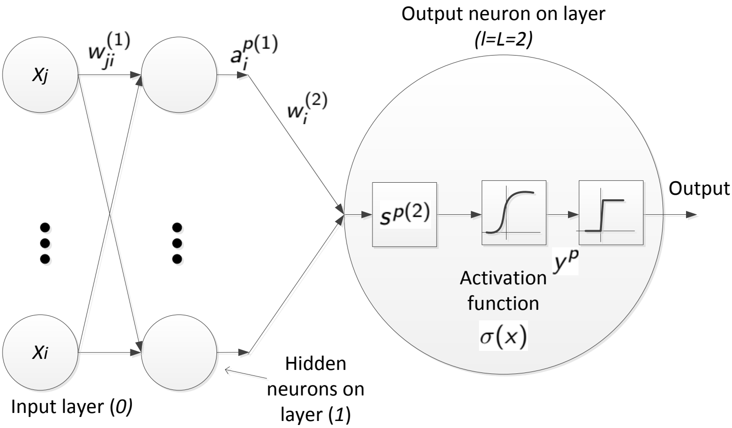
$$= a_i'^{(l-1)} \sum_{j=1}^{J^{(l)}} \delta_j^{(l)} \times w_{ij}^{(l)}$$



A simple 3-layer network

- We will use a simple 3-layer network
  - Output layer: $l = L = 2$. For simplicity, we assume there is only one output neuron,
  - One hidden layer: $l = 1$, we assume there are $J^{(1)}$ hidden neurons
  - Input layer: $l = 0$

A simple 2-layer network



Gradient Descent Learning for Cross Entropy Cost Function

- We define the Cross Entropy Cost Function

$$E_{ce} = -\sum_{p} [t^p \log(y^p) + (1 - t^p) \log(1 - y^p)]$$

- Reminder: We use sigmoid/logistic activation function for output neurons:

$$y^p = f^{(2)}(x) = \sigma(x)$$

and

$$a'^{(2)}(x) = \sigma'(x) = y^p(1 - y^p)$$

Backpropogation algorithm: a simple 3-layer network

- **Classification**: E^p_{ce} is the cross entropy cost function for sample p:

$$\delta^{(2)} = \frac{\partial E^p_{ce}}{\partial s^{p(2)}} = \frac{\partial E^p_{ce}}{\partial y^p}\frac{\partial y^p}{\partial s^{p(2)}}$$

  where $y^p = \sigma(s_1^{p(2)})$

- Since:

$$\frac{\partial E^p_{ce}}{\partial y^p} = \frac{y^p - t^p}{y^p(1-y^p)}$$

  and

$$\frac{\partial y^p}{\partial s^{(2)}} = \sigma'(s^{(2)}) = y^p(1-y^p)$$

  we have

$$\delta^{p(2)} = \frac{\partial E^p_{ce}}{\partial s^{p(2)}} = y^p - t^p$$

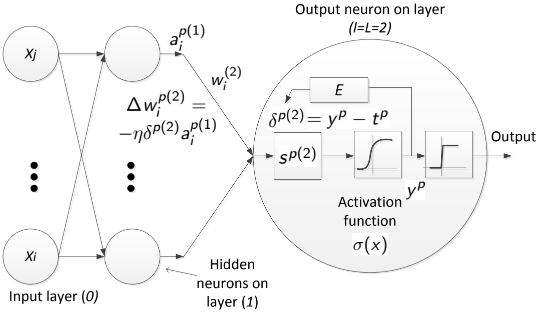
Backpropogation algorithm: a simple 3-layer network

The general weight update equation:

$$\Delta w_i^{p(2)} = -\eta \delta^{p(2)} a_i^{p(1)}$$

where $\delta^{p(2)} = y^p - t^p$

Note: please take a look at the weight update equation for single layer perceptron classification (Lecture 6 in Week 3)



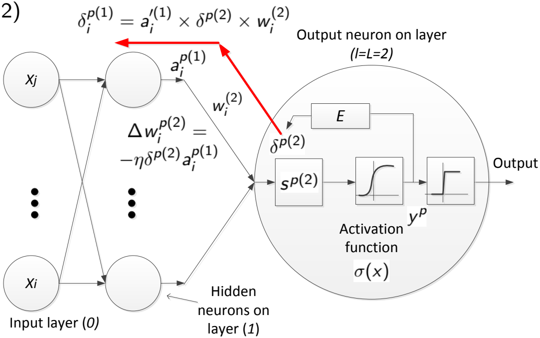
Backpropogation algorithm: δ_j for the hidden layers

- For neuron i on **hidden layer** 1:

$$\delta_i^{p(1)} = a_i'^{p(1)} \times \delta^{p(2)} \times w_i^{(2)}$$

- We use $\tanh(x)$ as activation function for hidden neurons: $a_i^{p(1)} = f(s_i^{p(1)}) = \tanh(s_i^{p(1)})$, and

$$a_i'^{p(1)} = 1 - (a_i^{p(1)})^2$$



Backpropogation algorithm: δ_j for the hidden layers

- For neuron i on **hidden layer** 1 and the weight connecting neurons j in previous layer:

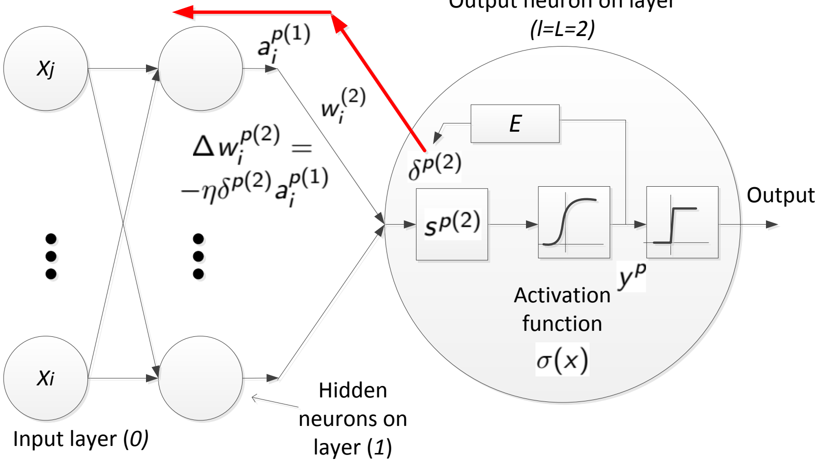
$$\Delta w_{ji}^{p(1)} = -\eta \delta_i^{p(1)} a_j^{p(0)}$$

- Since for a 2-layer MLP: $a_j^{p(0)} = x_j^p$, for neuron i on **hidden layer** 1:

$$\Delta w_{ji}^{p(1)} = -\eta a_i'^{p(1)} \delta^{p(2)} w_i^{(2)} x_j^p$$

Backpropogation algorithm: δ_j for the hidden layers

$$\Delta w_{ji}^{p(1)} = -\eta a_i'^{p(1)} \delta^{p(2)} w_i^{(2)} x_j^p$$



Putting them together: BP with stochastic gradient descent

Inputs: Training samples $\{\mathbf{x}, \mathbf{y}\}^p$, where $p = 1, \cdots, P$, $\mathbf{x} \in \mathbb{R}^M$ and $\mathbf{y} \in \mathbb{R}^N$
begin
  Generate random initial weights $w_{ij}^{(l)}$
  Randomly reshuffle training samples
  repeat
    for each training sample p :
      Forward: Compute all neuron activation function $a_j^{p(l)}$
      Backward: Compute all error signals $\delta_j^{p(l)}$
      Update the weights:
      $w_{ij}^{(l)}(t+1) = w_{ij}^{(l)}(t) - \eta a_i^{p(l-1)} \delta_j^{p(l)}$
    end for
  until $E(\mathbf{w}) < \epsilon$
end

Pyton Implementation to solve a classification problem

- The problem: a binary non-linearly separable classification problem
- Using make_moon function to generate 2d binary classification problems where data points are two interleaving half circles