Back-Propagation Algorithm

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

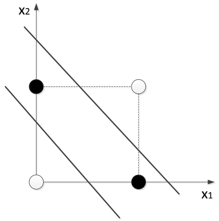
Single layer network vs multilayer networks

- Drawback of single layer Perceptrons: cannot deal with non-linearly separable problems like XOR
- How to solve this simple binary classification using neural networks.

Table : Logic gate: XOR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Figure : Decision Boundary of XOR

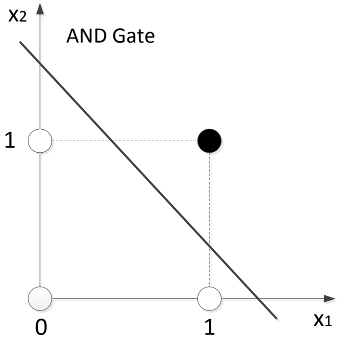


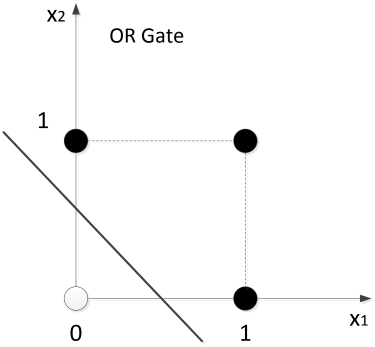
Solving XOR problem using ANNs

- How to solve this simple non-linearly separable classification problem using neural networks.
- We need three things to define a neural networks:
  - **Network topology:** to define how neurons are connected by weights
  - **Activation function:** to convert a neuron's weighted input to its output activation
  - **Learning process:** to update the weights

Solving XOR problem using ANNs

Question: how to design a ANN to solve XOR problem?
Hint:



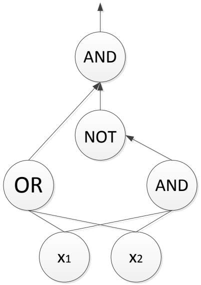


The need for Multiple layers

- From digital logic gate we know ($\oplus$ means XOR):

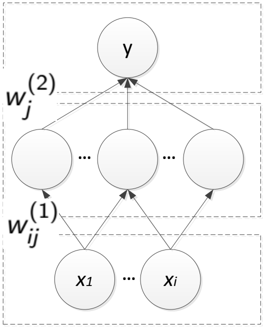
  $$x_1 \oplus x_2 = (x_1 + x_2) \cdot \overline{(x_1 \cdot x_2)}$$

- **Network topology:** we need Multi-Layer Percetrons (MLPs)



Notation for Multi-Layer Networks

- We use $a_j^{(l)}$ to denote the activation (output value) of unit j in layer l, where $l = 0, \cdots, L$, and $f^{(l)}(x)$ is its activation function. $w_{ij}^{(l)}$ are the weights in layer l.



Output layer (n=2)

$$a_1^{(2)} = y = f^{(2)}\left(\sum_j a_j^{(1)} w_j^{(2)}\right)$$

Hidden layer (n=1)

$$a_j^{(1)} = f^{(2)}\left(\sum_i a_i^{(0)} w_{ij}^{(1)}\right)$$

Input layer (n=0)

$$a_i^{(0)} = x_i$$

Activation function: Question

- **Recall**: For the regression problem we used linear output activation functions, i.e., $y = f^{(l)}(x) = x$
- **Question**: can we use these linear activation functions on the hidden layers to solve **general** non-linearly separable problems?

Answer: The Need For Non-Linearity

- **Question**: can we use these linear activation functions on the hidden layers to solve **general** non-linearly separable problems?
- **Answer**: No. Assuming the activation functions of the hidden layer of a 2-layer MLP are $f^{(1)}(x) = x$, we have

$$y = f^{(2)}\left(\sum_j a_j^{(1)} w_j^{(2)}\right) = f^{(2)}\left(\sum_j f^{(1)}\left(\sum_i x_i w_{ij}^{(1)}\right) w_j^{(2)}\right)$$

$$= f^{(2)}\left(\sum_j \sum_i x_i w_{ij}^{(1)} w_j^{(2)}\right) = f^{(2)}\left(\sum_i x_i \sum_j w_{ij}^{(1)} w_j^{(2)}\right)$$

which is equivalent to a single layer network with weights $w_i = \sum_j w_{ij}^{(1)} w_j^{(2)}$

Non-linear Activation/Transfer Functions

- Logistic sigmoid function
- An alternatives:
  - Hyperbolic tangent, which is related to the standard logistic function:
    $$f(x) = \tanh(x) = 2\sigma(2x) - 1$$
  - Its derivative:
    $$f'(x) = 1 - f(x)^2$$
  - Because some empirical evidence show that this function enables the gradient descent algorithm to learn faster, it has been widely used in MLP as hidden layer activation function.

Learning in Multi-Layer Perceptrons

- **Principle**: Training N-layer neural networks follows the same principle as for single layer networks, i.e., adjusting weight $\mathbf{w}$ to minimize an output cost function
  - Regression:

$$E_{sse}(\mathbf{w}) = \sum_{p=1}^{P}(t^p - y^p)^2$$

  - Classification:

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

  - We minimise them by a series of gradient descent weight updates:

$$\Delta \mathbf{w}^{(m)} = -\eta \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}^{(m)}}$$

Back propagation algorithm: Forward step

Prerequisite for Back propagation

- **Forward step**: to get y to calculate $E(\mathbf{w})$
  - **Essence**: propagating input layer by layer through the network to compute the final output of the network

How to update weights efficiently?

- We use brutal force to calculate all $\Delta w_{ij}^{(l)} = \frac{\partial E(\mathbf{w})}{\partial w_{ij}^{(l)}}$ one by one
- How to update weights efficiently?
- Back-propagation: the most wi(l)dly used learning algorithm for MLP
- Intuition: Use the neural network at the current state (topology and weights) and inputs to produce outputs, measure the errors and propagated back through the network to provide a signal to update weights.

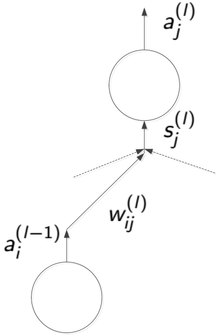
House keeping staff: new definition

- 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:

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



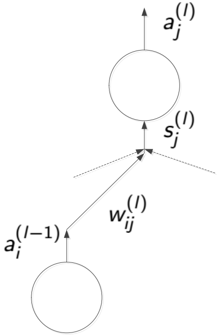
How to update weights efficiently?

- Using the chain rule:

$$\Delta w_{ij}^{(l)} = -\eta \frac{\partial E(\mathbf{w})}{\partial w_{ij}^{(l)}} = -\eta \frac{\partial E(\mathbf{w})}{\partial s_j^{(l)}} \times \frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}}$$

- We have

$$\frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}} = \frac{\partial \sum_i w_{ij}^{(l)} a_i^{(l-1)}}{\partial w_{ij}^{(l)}} = a_i^{(l-1)}$$



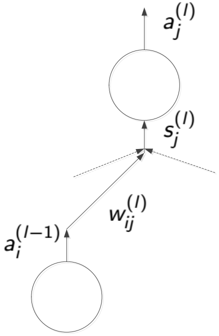
How to update weights efficiently?

- We only need:

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

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

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



Backpropogation algorithm: δ_j for the output layer

- Using the forward step we can get y to calculate $E(\mathbf{w})$
- For neuron j on the output (final) layer $l = L$:

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

- $E(\mathbf{w})$ is the output cost function, let's consider regression:

$$E_{sse}(\mathbf{w})^p = \frac{1}{2}\sum_{p=1}^{P}(y^p - t^p)^2$$

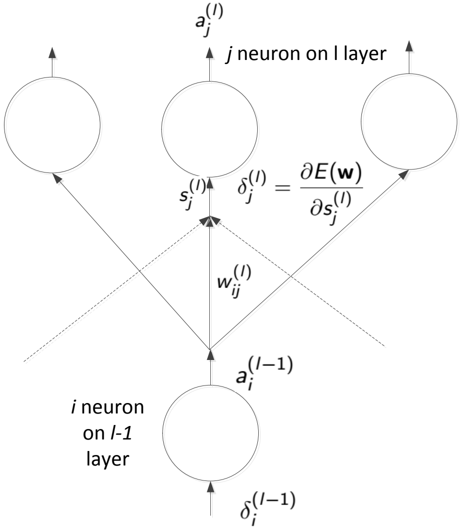
and

$$y^p = s_j^{(L)}$$

- We get:

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

Backpropogation algorithm: δ_j for any hidden layers


$a_j^{(l)}$
j neuron on l layer
$s_j^{(l)}$
$\delta_j^{(l)} = \frac{\partial E(\mathbf{w})}{\partial s_j^{(l)}}$
$w_{ij}^{(l)}$
$a_i^{(l-1)}$
i neuron on *l-1* layer
$\delta_i^{(l-1)}$

Backpropogation algorithm: δ_j for the hidden layers

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

$$\begin{aligned}\delta_i^{(l-1)} &= \frac{\partial E(\mathbf{w})}{\partial s_i^{(l-1)}} \\ &= \sum_{j=1}^{J^{(l)}} \frac{\partial E(\mathbf{w})}{\partial s_j^{(l)}} \times \frac{\partial s_j^{(l)}}{\partial a_i^{(l-1)}} \times \frac{\partial a_i^{(l-1)}}{\partial s_i^{(l-1)}} \\ &= \sum_{j=1}^{J^{(l)}} \delta_j^{(l)} \times w_{ij}^{(l)} \times f'(s_i^{(l-1)})\end{aligned}$$

- If the activation function is the $f(x) = \tanh(s_i^{(l-1)})$, we have

$$\delta_i^{(l-1)} = (1 - (a_i^{(l-1)})^2) \sum_{j=1}^{J^{(l)}} w_{ij}^{(l)} \times \delta_j^{(l)}$$

Backpropogation algorithm: δ_j for the hidden layers

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

$$\Delta w_{ij}^{(l-1)} = -\eta \frac{\partial E(\mathbf{w})}{\partial w_{ij}^{(l-1)}} = -\eta \delta_j^{(l-1)} a_i^{(l-2)}$$

$$= -\eta a_i^{(l-2)} (1 - (a_i^{(l-1)})^2) \sum_{j=1}^{J^{(l)}} w_{ij}^{(l)} \times \delta_j^{(l)}$$

- A 2-layer MLP: $a_i^{(l-2)} = a_i^{(0)} = x_i$

$$\Delta w_{ij}^{(l-1)} = -\eta x_i (1 - (a_i^{(l-1)})^2) \sum_{j=1}^{J^{(l)}} w_{ij}^{(l)} \times \delta_j^{(l)}$$

Drawbacks of Back-propogation

- Inefficient locking, i.e., during back-propogation of one layer, all other layers must wait for the that layer to execute forwards and propagate error backwards before they can be updated
  Decoupled Neural Interfaces using Synthetic Gradients