Machine Learning: Algorithms and Applications

Advanced Multimedia Research Lab
University of Wollongong

Artificial Neural Networks and Deep Learning: An Introduction (I)

Neural networks

- A neural network is a machine designed to model the way in which the brain performs a particular task or function of interest.
- A neural network is a massively parallel distributed processor made up of simple processing units that has a natural propensity for storing experiential knowledge and making it available for use haykin2009.

It resembles the brain in two respects haykin2009:

1. Knowledge is acquired by the network from its environment through a learning process.
2. Inter-neuron connection strengths, known as synaptic weights, are used to store the acquired knowledge.

Models of a neuron

- A neuron is an information-processing unit fundamental to the operation of a neural network
- Consists of:
  1. Synapse or connecting links: each characterized by a weight (ω_{kj}) or strength of its own. Note a signal x_j at the input of synapse j, connected to neuron k is multiplied by the synaptic weight ω_{kj}.
  2. Adder: sums the input signals(x_i), weighted by the respective synaptic strengths of the neuron
  3. Activation (or squashing) function: limits the amplitude of the output of a neuron; squashes permissible amplitude range of the output signal to some finite value.

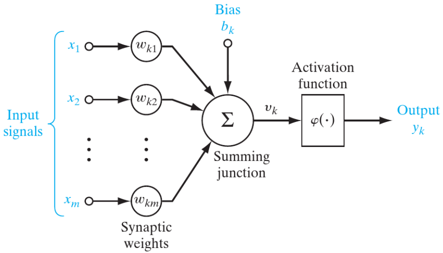


Figure 1: Model of a neuron with bias b_k which increases or lowers the net input of the activation function haykin2009.

Models of a neuron

Operation of neuron in Figure (**1**) can be written mathematically as

$$u_k = \sum_{j=1}^{m} \omega_{kj} x_j \tag{1}$$

$$y_k = \varphi(u_k + b_k) \tag{2}$$

where

- $x_1, x_2, \ldots x_m$ are the input signals;
- $\omega_1, \omega_2, \ldots, \omega_m$ are the respective synaptic weights of neuron k;
- u_k is the linear combiner output due to the input signals
- b_k is the bias;
- $\varphi(\cdot)$ is the activation function;

Bias b_k applies an affine transformation to the output u_k of the linear combiner

$$v_k = u_k + b_k \tag{3}$$

Equations (**1**) - (**3**) can be combined into

$$v_k = \sum_{j=0}^{m} \omega_{kj} x_j \qquad (4)$$

and

$$y_k = \varphi(v_k) \qquad (5)$$

In combining the equations a new synapse has been added with input

$$x_0 = +1 \qquad (6)$$

and weight

$$\omega_{k0} = b_k \qquad (7)$$

See Figure (**2**).

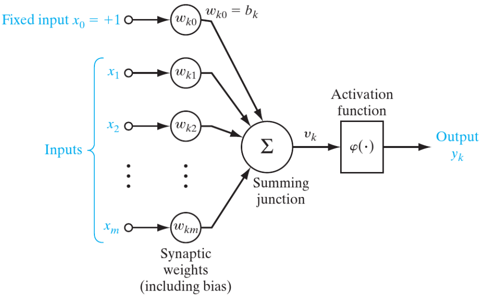
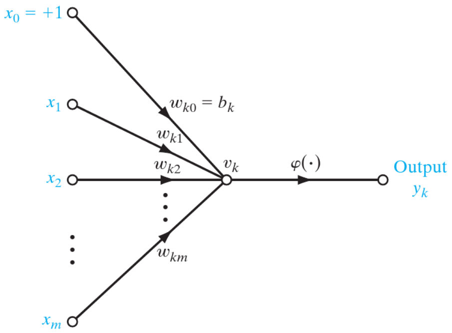


Figure 2: Model of neuron with the bias absorbed into the neuron haykin2009.

Models of a neuron

- Signal flow model of a neuron could be useful in some analysis or visualization
- Output is given by Equations (**4**) & (**5**)



FI

Figure 3: Signal flow model of a neuron haykin2009

Common Activation Functions

Threshold Function depicted in Figure (**3**) can be written as:

$$\varphi(v) = \begin{cases} 1 & \text{if} \quad v \geq 0 \\ 0 & \text{if} \quad v < 0 \end{cases} \tag{8}$$

Output of neuron, k, using threshold function is

$$y_k = \begin{cases} 1 & \text{if} \quad v_k \geq 0 \\ 0 & \text{if} \quad v_k < 0 \end{cases} \tag{9}$$

and induced local field of neuron, v_k is

$$v_k = \sum_{j=1}^{m} \omega_{kj} x_j + b_k \tag{10}$$

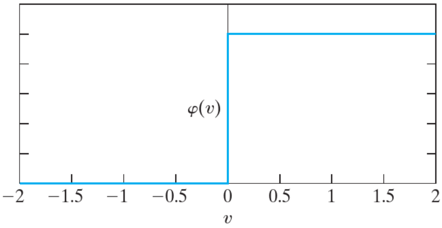


Figure 4: Threshold function haykin2009.

Common Activation Functions

Logistic Function (an example of Sigmoid function) is depicted in Figure (**5**) and can be written as:

$$\varphi(v) = \frac{1}{1 + \exp(-av)} \qquad (11)$$

where induced local field of neuron, v_k is

$$v_k = \sum_{j=1}^{m} \omega_{kj} x_j + b_k \qquad (12)$$

and slope parameter a determines the shape

- Note that the logistic function is differentiable while the threshold function is not

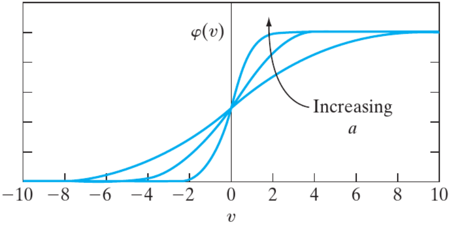


Figure 5: Sigmoid function for varying slope parameter a haykin2009.

Common Activation Functions

- Rectified Linear Unit (ReLU) has become very popular since its introduction by (**1**).
- Output is a non-linear function of the input

$$v_k = \sum_{j=1}^{m} \omega_{kj} x_j + b_k \tag{13}$$

$$y_k = \begin{cases} v_k & \text{if} \quad v_k > 0 \\ 0 & \text{if} \quad v_k < 0 \end{cases} \tag{14}$$

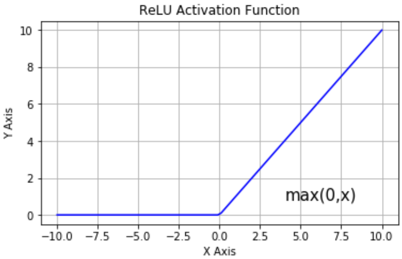


Figure 6: Rectified Linear Unit

Common Activation Functions

- Softmax activation function squashes each input to a value between 0 and 1.
- Output is equivalent to a categorical probability distribution
- Graph similar to logistic but usually applied to provide probabilistic interpretation to outputs in classification task

$$v_k = \sum_{j=1}^{m} \omega_{kj} x_j + b_k \tag{15}$$

$$y_k = \frac{\exp(v_k)}{\sum_{k=1}^{K} \exp(v_k)} \tag{16}$$

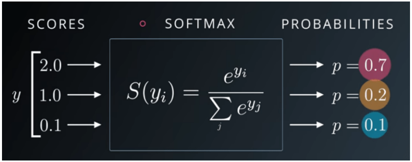


Figure 7: Softmax operation for a 3-class classification task (`https://sefiks.com/`).

Common Activation Functions

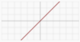
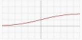
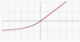
Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

Figure 8: Activation Functions (`https://towardsdatasience.com`)

Common Activation Functions

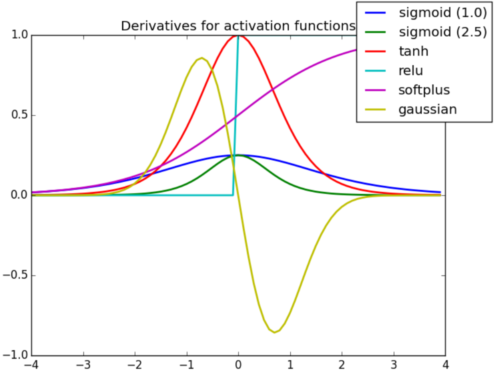


Figure 9: Derivative of Activation Functions (`https://towardsdatasience.com`)

Common Activation Functions

What are some nice properties of activation functions?

- Nonlinear function; otherwise neural net can only solve simple problems;
- Without activation neural net is equivalent to a linear regression
- Nice derivatives makes learning easy
- Activation functions should give a bounded output for a bounded input
- Choosing the right activation function is both science and art. For further insight, see the works of (**1**) and (**2**)
- Together with the right cost function, activation functions make training NN possible.

Models of a neuron

- In Figure (**10**) consider only 3 inputs and the bias into the neuron;
- Let the weights be $\omega_{10} = b_1 = 0.5$, $\omega_{11} = 0.4$ $\omega_{12} = 0.6$; $\omega_{13} = 0.2$
- Let the inputs be $x_0 = 1$; $x_1 = 1.2$; $x_2 = 2.0$; $x_3 = 1.8$
- Let the activation function be logistic sigmod with $a = 0.2$

$$\begin{aligned} v_1 &= \sum_{j=0}^{3} \omega_{1j} x_j \\ &= 1 \times 0.5 + 0.4 \times 1.2 + 0.6 \times 2.0 + 0.2 \times 1.8 \\ &= 2.54 \end{aligned}$$

$$\begin{aligned} y_1 = \varphi(v_1) &= \frac{1}{1 + \exp(-av_1)} \\ &= \frac{1}{1 + \exp(-0.2 \times 2.54)} = 0.624 \end{aligned}$$

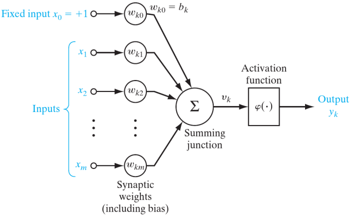


Figure 10: Model of neuron: Example computation haykin2009.

Network Architecture

Single Layer Feedforward Networks

- Input layer of source nodes project directly onto an output layer of neurons

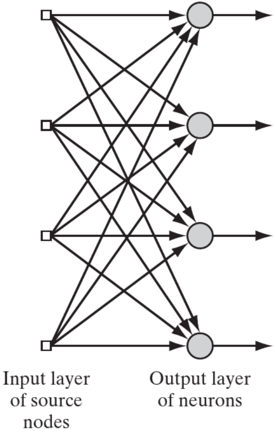


Figure 11: Single Layer Feedforward NN ((**?**))

Network Architecture

Multilayer Feedforward Networks

- Input layer of source nodes project directly onto a set of neurons in a hidden layer
- There could be one or more hidden layers; output of each layer forming input to the next layer
- Adding one or more hidden layers allows network to extract higher-order statistics from the input data
- Network is fully connected if every node in each layer is connected to every node in the adjacent forward layer

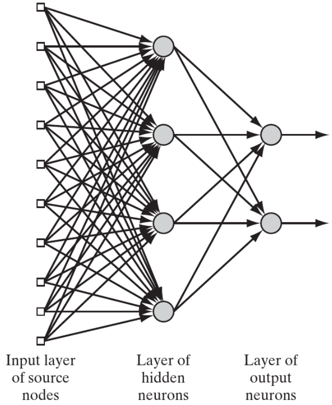


Figure 12: Multilayer Fully Connected Feedforward NN ((**?**))

Network Architecture

Recurrent Networks

- Unlike feedforward networks recurrent networks introduce feedback from output to input and with multilayer feedback could also be among layers
- Feedback loops and nonlinear activation functions allow neural network to model nonlinear dynamic systems

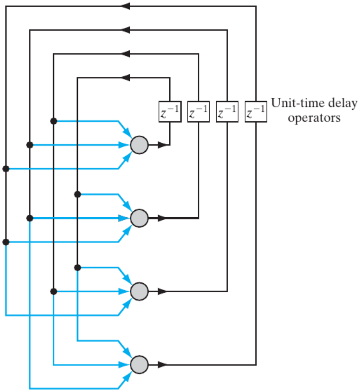


Figure 13: Single Layer Recurrent Neural Network ((**?**))

Network Architecture

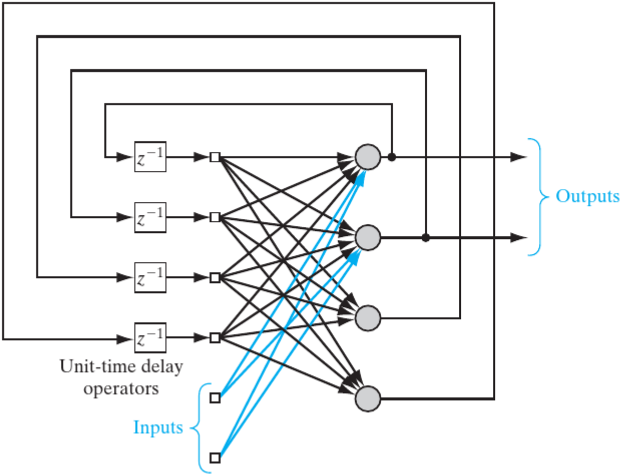


Figure 14: Recurrent Neural Network with Hidden Layer ((**?**))

Learning process

Types of Learning

- Supervised learning - predict an output when given an input vector
- Reinforcement learning - select an action to maximize some defined payoff
- Unsupervised learning - discover a good internal representation of the data

Learning process

Supervised Learning

- Each training case consists of an input vector x and a target output t.
  1. Regression: The target output is a real number or a whole vector of real numbers.
  2. Classification: The target output is a class label.

Recall that in general we want to learn a mapping from input vector $\boldsymbol{x}$ to some output y through a vector of weights $\boldsymbol{\omega}$

$$y = f(\boldsymbol{\omega}, \boldsymbol{x}) \tag{17}$$

such that the error (or loss or cost function) incurred in the prediction of the actual value is minimized.

- For regression, the cost function

$$J(\boldsymbol{\omega}, b) = -\mathbb{E} \log p_{\text{model}}(y|\boldsymbol{x}) \tag{18}$$

is the expectation of negative conditional log-likelihood computed over the training data; the cross-entropy between the training data and the model distribution

Learning process

- Cost function in Equation (**18**) is usually minimized in an optimization process, gradient descent.
- How to understand gradient-based optimization? (
  - Consider a function $y = f(x)$ where both x and y are real numbers
  - Derivative of $y = f(x)$, $f'(x)$, gives slope of $f(x)$ at point x
  - Importantly, it tells us how to scale a small change in the input to obtain corresponding change in output (this is due to Taylor's expansion):

$$f(x+\epsilon) \approx f(x) + \epsilon f'(x) \tag{19}$$

$$f(x - \epsilon \,\text{sign}(f'(x))) < f(x) \quad \text{for small enough} \quad \epsilon$$

    So we reduce $f(x)$ by moving x in small steps with the opposite sign of the derivative

  - This technique is called gradient descent [1] and credited to Louis Augustin Cauchy, 1847 (it's also called steepest descent)

[1] For brief (mathematical) historical account see (**?**)

Learning process

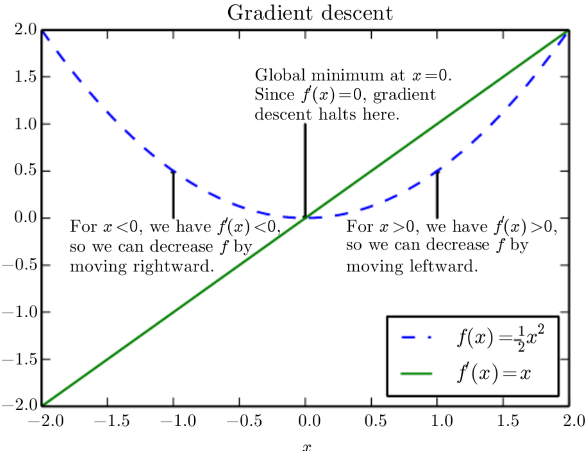


Figure 15: Illustraion of the gradient descent algorithm (**?** p.80)

Learning process

- In general the input to the function f is a vector $\boldsymbol{x}$, so we consider generalization of the derivative of f, ∇f
- Let $\boldsymbol{x} = \{x_1, x_2, \ldots x_m\}$;

$$\nabla f(\boldsymbol{x}) = \left[\frac{\partial f}{\partial x_1}, \quad \frac{\partial f}{\partial x_2} \ldots \quad \frac{\partial f}{\partial x_m}\right]^t$$

- Partial derivative $\dfrac{\partial f}{\partial x_i}$ measures how f changes as only the variable x_i increases at point $\boldsymbol{x}$.
- Directional derivative in the direction of a unit vector $\boldsymbol{u}$ is the slope of f in the direction of $\boldsymbol{u}$

Learning process

- Directional derivative is derivative of $f(\boldsymbol{x}+\alpha\boldsymbol{u})$ with respect to α evaluated at $\alpha = 0$
- Chain rule says that given a function $f(u)$, and $u(x)$; $\dfrac{\partial f}{\partial x} = \dfrac{\partial u}{\partial x}\dfrac{\partial f}{\partial u}$ therefore,

$$\frac{\partial}{\partial \alpha} f(\boldsymbol{x}+\alpha\boldsymbol{u}) = \boldsymbol{u}^t \nabla f(\boldsymbol{x}) = ||\boldsymbol{u}||_2 ||\nabla f(\boldsymbol{x})||_2 \cos\theta$$

- Minimize f by finding the direction in which f decreases fastest; Do this by minimizing the directional derivative

$$\min_{\boldsymbol{u},\boldsymbol{u}^t\boldsymbol{u}=1} \boldsymbol{u}^t \nabla f(\boldsymbol{x}) = \min_{\boldsymbol{u},\boldsymbol{u}^t\boldsymbol{u}=1} ||\boldsymbol{u}||_2 ||\nabla f(\boldsymbol{x})||_2 \cos\theta$$

  Minimum is achieved when $\boldsymbol{u}$ points in the opposite direction to $\nabla f(\boldsymbol{x})$

- We can decrease f by moving in the direction of negative gradient, choosing a new point as

$$\boldsymbol{x}' = \boldsymbol{x} - \epsilon \nabla f(\boldsymbol{x}); \quad \text{where } \epsilon \text{ is step size} \tag{20}$$

Perceptron

- Consider the perceptron shown in Figure (**16**); weights $\omega_i; i = \{1, \ldots m\}$; inputs $x_i; i = \{1, \ldots m\}$; external bias, b
- Correctly classify externally applied inputs into two classes $\mathcal{C}_1$ or $\mathcal{C}_2$
- If $y = +1$ classify to class $\mathcal{C}_1$; if $y = -1$ classify to $\mathcal{C}_2$

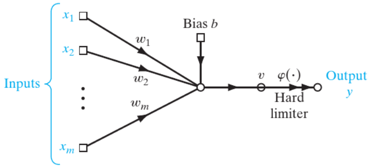


Figure 16: Signal flow model of the perceptron (**?**)

Perceptron

- Simple perceptron creates a hyperplane separating the two regions (see Figure(**17**))

$$\sum_{i=1}^{m} \omega_i x_i + b = 0$$

- Weights of perceptron adapted at each iteration of training sample presentation
- Use error-correction rule - perceptron convergence algorithm

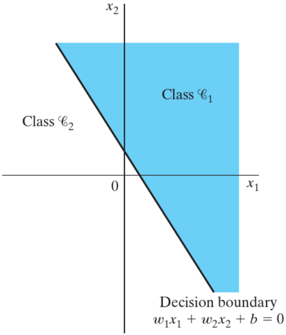


Figure 17: Hyperplane as decision boundary of 2-D, 2-class classification (**?**)

Perceptron

- The output of the linear combiner at iteration n, can be written as

$$v(n) = \sum_{i=0}^{m} \omega_i x_i = \boldsymbol{w}^t(n)\boldsymbol{x}(n)$$

- Classes $\mathcal{C}_1$ and $\mathcal{C}_2$ must be linearly separable for the perceptron to function properly (See Figure(**17**))
- Algorithm:

$$\boldsymbol{\omega}(n+1) = \begin{cases} \boldsymbol{\omega}(n) & \text{if} \quad \boldsymbol{w}^t(n)\boldsymbol{x}(n) > 0 \quad \text{and} \quad \boldsymbol{x}(n) \in \mathcal{C}_1 \\ \boldsymbol{\omega}(n) & \text{if} \quad \boldsymbol{w}^t(n)\boldsymbol{x}(n) \leq 0 \quad \text{and} \quad \boldsymbol{x}(n) \in \mathcal{C}_2 \end{cases}$$

otherwise

$$\boldsymbol{\omega}(n+1) = \begin{cases} \boldsymbol{\omega}(n) - \eta(n)\boldsymbol{x}(n) & \text{if} \quad \boldsymbol{w}^t(n)\boldsymbol{x}(n) > 0 \quad \text{and} \quad \boldsymbol{x}(n) \in \mathcal{C}_2 \\ \boldsymbol{\omega}(n) + \eta(n)\boldsymbol{x}(n) & \text{if} \quad \boldsymbol{w}^t(n)\boldsymbol{x}(n) \leq 0 \quad \text{and} \quad \boldsymbol{x}(n) \in \mathcal{C}_1 \end{cases}$$

Perceptron

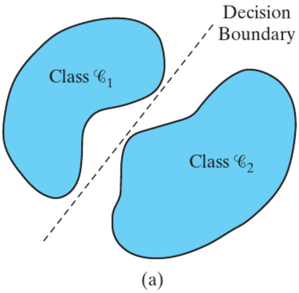


Figure 18: (a) Linearly separable patterns; (b) Linearly non-separable patterns (**?**)

Multilayer Perceptron

Basic features of multilayer perceptrons haykin2009 (See Figure **19**):

- Each neuron in the network includes a nonlinear activation function that is differentiable
- Network contains one or more layers that are hidden from both the input and output nodes
- Network exhibits a high degree of connectivity determined by synaptic weights of the network

Training method

Multilayer perceptron is usually trained using the back-propagation algorithm:

- Forward phase: Weights of the network are fixed and input signal is propagated layer-wise through the network and transformed signal appears at the output
- Backward phase: Error signal is computed by comparing generated output and desired response; error signal is propagated backward and layer-wise through the network; successive adjustments made to weights of the network

Multilayer Perceptron

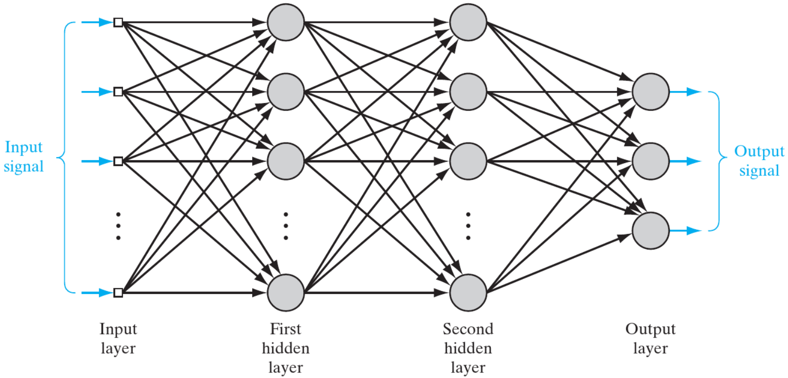


Figure 19: Architectural graph of the Multilayer Perceptron haykin2009

Multilayer Perceptron

- Each hidden or output neuron performs two computations:
  1. Output of each neuron expressed as continuous nonlinear function of input signals and associated weights
  2. Estimate of the gradient vector (gradient of error surface) required in the backward phase of the training
- Hidden neurons act as feature detectors, discovering the salient features characterising the training data;
- Hidden neurons perform nonlinear transformation on input data into a new space; feature space
- The training is a form of error-correction learning that assigns blame or credit to each of the internal neurons; this is a case of the credit assignment problem
- Back-propagation solves the credit assignment problem for the multilayer perceptron

Back-propagation Algorithm

Key points leading to overall strategy

- Multilayer perceptron is a universal function approximator
- It can be trained using error-correction learning to obtain optimum approximation
- The optimum can be obtained if we can minimize the approximation error
- This is equivalent to modifying the weights so that the network minimizes the error between desired output and response of the network
- Gradient descent algorithm can be used to find the minimum of an objective function by iteratively computing the adjustment that leads to the minimization of the objective function
- Back-propagation is an efficient implementation of the gradient descent
- Strategy is to compute the adjustment, $\Delta\omega$ to be applied to each weight, ω
- From Equation (**20**) the adjustment is proportional to the gradient of the objective function; in this case ∇E (E is error signal energy) with respect to the parameters ω

Back-propagation Algorithm

- Error signal of the output neuron is given by

$$e_j(n) = d_j(n) - y_j(n) \tag{21}$$

where y_j is the output of neuron j when stimulus $\boldsymbol{x}(n)$ is applied at the input; $d_j(n)$ is the desired output

- Instantaneous error energy can be written as

$$E_j(n) = \frac{1}{2} e_j^2(n) \tag{22}$$

- Total instantaneous error (summed over all neurons in the output layer) is

$$E(n) = \sum_{j \in C} E_j(n) = \frac{1}{2} \sum_{j \in C} e_j^2(n) \tag{23}$$

- Computation of the error could be in batch mode or on-line mode leading to either batch mode (presentation of all training samples) or on-line (presentation of training sample one-at-a-time) training

Back-propagation Algorithm

- Error signal of the output neuron is given by

$$e_j(n) = d_j(n) - y_j(n) \tag{21}$$

where y_j is the output of neuron j when stimulus $\boldsymbol{x}(n)$ is applied at the input; $d_j(n)$ is the desired output

- Instantaneous error energy can be written as

$$E_j(n) = \frac{1}{2} e_j^2(n) \tag{22}$$

- Total instantaneous error (summed over all neurons in the output layer) is

$$E(n) = \sum_{j \in C} E_j(n) = \frac{1}{2} \sum_{j \in C} e_j^2(n) \tag{23}$$

- Computation of the error could be in batch mode or on-line mode leading to either batch mode (presentation of all training samples) or on-line (presentation of training sample one-at-a-time) training

Back-propagation Algorithm

- Error signal of the output neuron is given by

$$e_j(n) = d_j(n) - y_j(n) \tag{21}$$

  where y_j is the output of neuron j when stimulus $\boldsymbol{x}(n)$ is applied at the input; $d_j(n)$ is the desired output

- Instantaneous error energy can be written as

$$E_j(n) = \frac{1}{2} e_j^2(n) \tag{22}$$

- Total instantaneous error (summed over all neurons in the output layer) is

$$E(n) = \sum_{j \in C} E_j(n) = \frac{1}{2} \sum_{j \in C} e_j^2(n) \tag{23}$$

- Computation of the error could be in batch mode or on-line mode leading to either batch mode (presentation of all training samples) or on-line (presentation of training sample one-at-a-time) training

Back-propagation Algorithm

- Error signal of the output neuron is given by

$$e_j(n) = d_j(n) - y_j(n) \tag{21}$$

where y_j is the output of neuron j when stimulus $\boldsymbol{x}(n)$ is applied at the input; $d_j(n)$ is the desired output

- Instantaneous error energy can be written as

$$E_j(n) = \frac{1}{2} e_j^2(n) \tag{22}$$

- Total instantaneous error (summed over all neurons in the output layer) is

$$E(n) = \sum_{j \in C} E_j(n) = \frac{1}{2} \sum_{j \in C} e_j^2(n) \tag{23}$$

- Computation of the error could be in batch mode or on-line mode leading to either batch mode (presentation of all training samples) or on-line (presentation of training sample one-at-a-time) training

Consider Figure (**20**):

- Induced local field of neuron j at iteration n is:

$$v_j(n) = \sum_{i=0}^{m} \omega_{ji}(n) y_i(n) \tag{24}$$

  m is the total number of inputs

- Function signal $y_j(n)$ appearing at the output of neuron j at iteration n is

$$y_j(n) = \varphi_j(v_j(n)) \tag{25}$$

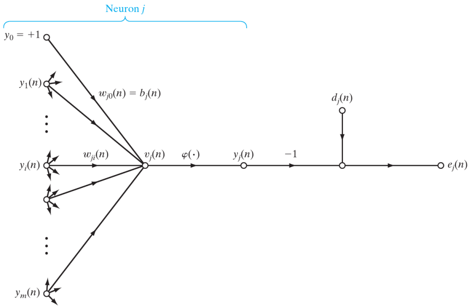


Figure 20: Signal flow highlighting neuron j being fed by the outputs from the neurons to its left; induced local field of neuron is $v_j(n)$ and this is the input to activation function $\varphi(\cdot)$ haykin2009

Back-propagation Algorithm

Consider Figure (**20**):

- Induced local field of neuron j at iteration n is:

$$v_j(n) = \sum_{i=0}^{m} \omega_{ji}(n) y_i(n) \tag{24}$$

  m is the total number of inputs

- Function signal $y_j(n)$ appearing at the output of neuron j at iteration n is

$$y_j(n) = \varphi_j(v_j(n)) \tag{25}$$

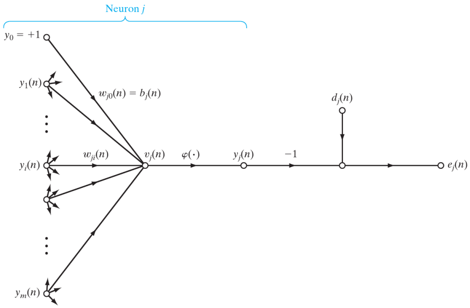


Figure 20: Signal flow highlighting neuron j being fed by the outputs from the neurons to its left; induced local field of neuron is $v_j(n)$ and this is the input to activation function $\varphi(\cdot)$ haykin2009

Back-propagation Algorithm

- We need to compute the adjustment (or correction) $\Delta\omega_{ji}(n)$ to be applied to weight $\omega_{ji}(n)$

- This is proportional to the partial derivative $\dfrac{\partial E(n)}{\partial \omega_{ji}(n)}$ and determines the direction of search in the weight space for ω_{ji}

- Chain rule tells us how to compute $\dfrac{\partial E(n)}{\partial \omega_{ji}(n)}$ from a set of known quantities

$$\frac{\partial E(n)}{\partial \omega_{ji}(n)} = \frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial \omega_{ji}(n)} \tag{26}$$

- Recall Equation (**??**) : $E_j(n) = \frac{1}{2}e_j^2(n)$; therefore

$$\frac{\partial E(n)}{\partial e_j(n)} = e_j(n) \tag{27}$$

Back-propagation Algorithm

- We need to compute the adjustment (or correction) $\Delta\omega_{ji}(n)$ to be applied to weight $\omega_{ji}(n)$
- This is proportional to the partial derivative $\frac{\partial E(n)}{\partial \omega_{ji}(n)}$ and determines the direction of search in the weight space for ω_{ji}
- Chain rule tells us how to compute $\frac{\partial E(n)}{\partial \omega_{ji}(n)}$ from a set of known quantities

$$\frac{\partial E(n)}{\partial \omega_{ji}(n)} = \frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial \omega_{ji}(n)} \tag{26}$$

- Recall Equation (**??**) : $E_j(n) = \frac{1}{2}e_j^2(n)$; therefore

$$\frac{\partial E(n)}{\partial e_j(n)} = e_j(n) \tag{27}$$

- We need to compute the adjustment (or correction) $\Delta\omega_{ji}(n)$ to be applied to weight $\omega_{ji}(n)$
- This is proportional to the partial derivative $\dfrac{\partial E(n)}{\partial \omega_{ji}(n)}$ and determines the direction of search in the weight space for ω_{ji}
- Chain rule tells us how to compute $\dfrac{\partial E(n)}{\partial \omega_{ji}(n)}$ from a set of known quantities

$$\frac{\partial E(n)}{\partial \omega_{ji}(n)} = \frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial \omega_{ji}(n)} \tag{26}$$

- Recall Equation (**??**) : $E_j(n) = \frac{1}{2}e_j^2(n)$; therefore

$$\frac{\partial E(n)}{\partial e_j(n)} = e_j(n) \tag{27}$$

Back-propagation Algorithm

- We need to compute the adjustment (or correction) $\Delta\omega_{ji}(n)$ to be applied to weight $\omega_{ji}(n)$
- This is proportional to the partial derivative $\dfrac{\partial E(n)}{\partial \omega_{ji}(n)}$ and determines the direction of search in the weight space for ω_{ji}
- Chain rule tells us how to compute $\dfrac{\partial E(n)}{\partial \omega_{ji}(n)}$ from a set of known quantities

$$\frac{\partial E(n)}{\partial \omega_{ji}(n)} = \frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial \omega_{ji}(n)} \tag{26}$$

- Recall Equation (**22**) : $E_j(n) = \frac{1}{2}e_j^2(n)$; therefore

$$\frac{\partial E(n)}{\partial e_j(n)} = e_j(n) \tag{27}$$

Back-propagation Algorithm

- Recall Equation (**21**): $e_j(n) = d_j(n) - y_j(n)$

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \tag{28}$$

- Recall Equation (**23**): $y_j(n) = \varphi_j(v_j(n))$

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n)); \text{ where } ()' \text{ indicates differentiation} \tag{29}$$

- Recall Equation(**??**): $v_j(n) = \sum_{i=0}^{m} \omega_{ji}(n) y_i(n)$

$$\frac{\partial v_j(n)}{\partial \omega_{ji}(n)} = y_i(n) \tag{30}$$

- Equation (**??**) becomes (using Equations (**??**) - (**??**))

$$\frac{\partial E(n)}{\partial \omega_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n) \tag{31}$$

Back-propagation Algorithm

- Recall Equation (**??**): $e_j(n) = d_j(n) - y_j(n)$

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \qquad (28)$$

- Recall Equation (**??**): $y_j(n) = \varphi_j(v_j(n))$

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n)); \text{ where } ()' \text{ indicates differentiation} \qquad (29)$$

- Recall Equation(**??**): $v_j(n) = \sum_{i=0}^{m} \omega_{ji}(n) y_i(n)$

$$\frac{\partial v_j(n)}{\partial \omega_{ji}(n)} = y_i(n) \qquad (30)$$

- Equation (**??**) becomes (using Equations (**??**) - (**??**))

$$\frac{\partial E(n)}{\partial \omega_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n) \qquad (31)$$

Back-propagation Algorithm

- Recall Equation (**??**): $e_j(n) = d_j(n) - y_j(n)$

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \tag{28}$$

- Recall Equation (**??**): $y_j(n) = \varphi_j(v_j(n))$

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n)); \text{ where } ()' \text{ indicates differentiation} \tag{29}$$

- Recall Equation(**??**): $v_j(n) = \sum_{i=0}^{m} \omega_{ji}(n) y_i(n)$

$$\frac{\partial v_j(n)}{\partial \omega_{ji}(n)} = y_i(n) \tag{30}$$

- Equation (**??**) becomes (using Equations (**??**) - (**??**))

$$\frac{\partial E(n)}{\partial \omega_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n) \tag{31}$$

Back-propagation Algorithm

- Recall Equation (**??**): $e_j(n) = d_j(n) - y_j(n)$

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \tag{28}$$

- Recall Equation (**??**): $y_j(n) = \varphi_j(v_j(n))$

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n)); \text{ where } ()' \text{ indicates differentiation} \tag{29}$$

- Recall Equation(**??**): $v_j(n) = \sum_{i=0}^{m} \omega_{ji}(n) y_i(n)$

$$\frac{\partial v_j(n)}{\partial \omega_{ji}(n)} = y_i(n) \tag{30}$$

- Equation (**??**) becomes (using Equations (**??**) - (**??**))

$$\frac{\partial E(n)}{\partial \omega_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n) \tag{31}$$

Back-propagation Algorithm

- Correction, $\Delta\omega_{ji}(n)$, applied to $\omega_{ji}(n)$ is defined by the delta rule

$$\begin{aligned}\Delta\omega_{ji}(n) &= -\eta\frac{\partial E(n)}{\partial\omega_{ji}(n)}; \quad \eta \quad \text{is the learning rate parameter} \\ &= \eta \boxed{e_j(n)\varphi_j'(v_j(n))}\, y_i(n) \\ &= \eta \boxed{\delta_j(n)}\, y_i(n) \end{aligned} \tag{32}$$

  where $\delta_j(n) = e_j(n)\varphi_j'(v_j(n))$ is defined as the local gradient for neuron j

- Local gradient for neuron j is the product of corresponding error $e_j(n)$ and the derivative of associated activation function, $\varphi_j'(v_j(n))$
- Error $e_j(n)$ is easily computed for the output neurons; we have access to $d_j(n)$ and $y_j(n)$. How to compute error for hidden neurons? These have no given $d_j(n)$.

Back-propagation Algorithm

- Correction, $\Delta\omega_{ji}(n)$, applied to $\omega_{ji}(n)$ is defined by the delta rule

$$\begin{aligned} \Delta\omega_{ji}(n) &= -\eta \frac{\partial E(n)}{\partial \omega_{ji}(n)}; \quad \eta \quad \text{is the learning rate parameter} \\ &= \eta \boxed{e_j(n)\varphi_j'(v_j(n))}\, y_i(n) \\ &= \eta \boxed{\delta_j(n)}\, y_i(n) \end{aligned} \tag{32}$$

  where $\delta_j(n) = e_j(n)\varphi_j'(v_j(n))$ is defined as the local gradient for neuron j

- Local gradient for neuron j is the product of corresponding error $e_j(n)$ and the derivative of associated activation function, $\varphi_j'(v_j(n))$

- Error $e_j(n)$ is easily computed for the output neurons; we have access to $d_j(n)$ and $y_j(n)$. How to compute error for hidden neurons? These have no given $d_j(n)$.

Back-propagation Algorithm

- Correction, $\Delta\omega_{ji}(n)$, applied to $\omega_{ji}(n)$ is defined by the delta rule

$$\begin{aligned}
\Delta\omega_{ji}(n) &= -\eta \frac{\partial E(n)}{\partial \omega_{ji}(n)}; \quad \eta \quad \text{is the learning rate parameter} \\
&= \eta \boxed{e_j(n)\varphi_j'(v_j(n))}\, y_i(n) \\
&= \eta \boxed{\delta_j(n)}\, y_i(n) \qquad (32)
\end{aligned}$$

  where $\delta_j(n) = e_j(n)\varphi_j'(v_j(n))$ is defined as the local gradient for neuron j

- Local gradient for neuron j is the product of corresponding error $e_j(n)$ and the derivative of associated activation function, $\varphi_j'(v_j(n))$
- Error $e_j(n)$ is easily computed for the output neurons; we have access to $d_j(n)$ and $y_j(n)$. How to compute error for hidden neurons? These have no given $d_j(n)$.

Back-propagation Algorithm

What do we know so far?

1. Training a multilayer perceptron involves using the training data set in an error-correction learning paradigm to adjust the weights
2. The error-correction learning is essentially equivalent to solving a function minimization problem
3. The function to be minimized is the error surface corresponding to the mismatch between the response of the network and the desired response
4. This can be solved by the gradient descent algorithm
5. The back-propagation algorithm is an efficient implementation of the gradient descent algorithm for the multilayer perceptron
6. The correction (or update) to the weight at each iteration is (cf. Equation (**??**)):

$$
\begin{aligned}
\Delta\omega_{ji}(n) &= \eta \boxed{e_j(n)\varphi_j'(v_j(n))}\, y_i(n) \\
&= \eta \boxed{\delta_j(n)}\, y_i(n)
\end{aligned}
\tag{33}
$$

This is the product of the learning rate η, local gradient of the associated neuron, $\delta_j(n)$ and the input to the neuron, $y_i(n)$. See Figure (**??**)

Back-propagation Algorithm

- Weights connected to the output neurons are updated as

$$\begin{aligned}\omega_{ji}^{new}(n) &= \omega_{ji}^{old}(n) + \Delta\omega_{ji}(n) \\ &= \omega_{ji}^{old}(n) + \eta \boxed{\delta_j(n)}\, y_i(n) \\ &= \omega_{ji}^{old}(n) + \eta \boxed{e_j(n)\varphi_j'(v_j(n))}\, y_i(n)\end{aligned} \tag{34}$$

- Using chain rule similarly to how we derive the update for the weight of output neurons we will show that the weight update for hidden neurons is given as

$$\begin{aligned}\omega_{ji}^{new}(n) &= \omega_{ji}^{old}(n) + \Delta\omega_{ji}(n) \\ &= \omega_{ji}^{old}(n) + \eta \boxed{\delta_j(n)}\, y_i(n) \\ &= \omega_{ji}^{old}(n) + \eta \boxed{\varphi_j'(v_j(n)) \sum_k \delta_k(n)\omega_{kj}(n)}\, y_i(n)\end{aligned} \tag{35}$$

where neuron j is hidden; $\varphi_j'(v_j(n))$ is derivative of associated activation function; $\delta_k(n)$ are associated with neurons k which are to the immediate right of neuron j and connected to it; $\omega_{kj}(n)$ are the associated weights of these connections (see Figure (**21**))

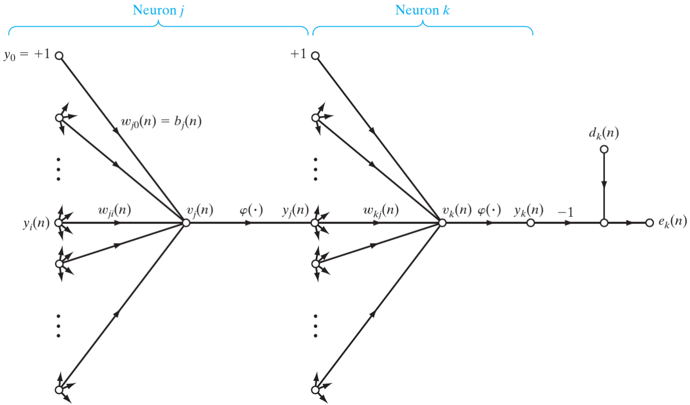


Figure 21: Signal flow showing hidden neuron j connected to an output neuron k to its immediate right; Diagram used to show the derivation of weight update for hidden neuron haykin2009

Back-propagation Algorithm

For the sake of completeness we now derive

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n)\omega_{kj}(n)$$

of Equation (**??**)

- Recall from Equation(**??**)

$$\frac{\partial E(n)}{\partial \omega_{ji}(n)} = \boxed{\frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)}} \frac{\partial v_j(n)}{\partial \omega_{ji}(n)}$$

  and Equation(**??**)

$$\begin{aligned} \Delta\omega_{ji}(n) &= \eta \boxed{e_j(n)\varphi_j'(v_j(n))} \, y_i(n) \\ &= \eta \boxed{\delta_j(n)} \, y_i(n) \end{aligned}$$

  we infer that the local gradient, $\delta_j(n)$, can be written as

$$\delta_j(n) = \frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \tag{36}$$

Back-propagation Algorithm

- Use Figure (**??**) and Equation (**??**) to write local gradient as:

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)}$$
$$= -\frac{\partial E(n)}{\partial y_j(n)} \varphi_j'(v_j(n)) \tag{37}$$

- From Figure (**??**)

$$E(n) = \frac{1}{2} \sum_{k \in C} e_k^2(n); \text{ neuron } k \text{ is an output node} \tag{38}$$

Differentiating both sides of Equation (**??**) with respect to y_j:

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_k e_k(n) \frac{\partial e_k(n)}{\partial y_j(n)} \tag{39}$$

Use chain rule to write

$$\frac{\partial e_k(n)}{\partial y_j(n)} = \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)}$$

and

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_k e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} \tag{40}$$

Back-propagation Algorithm

- Observe from Figure (**??**) that

$$
\begin{aligned}
e_k(n) &= d_k(n) - y_k(n) \\
&= d_k(n) - \varphi_k(v_k(n)); \text{ neuron } k \text{ is an output node} \qquad (41)
\end{aligned}
$$

  and we can write

$$
\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi_k'(v_k(n)) \qquad (42)
$$

- Also note that the induced local field for neuron k

$$
v_k(n) = \sum_{j=0}^{m} \omega_{kj}(n) y_j(n); \; m \text{ is number of inputs applied to neuron } k \qquad (43)
$$

  Upon differentiation we have

$$
\frac{\partial v_k(n)}{\partial y_j(n)} = \omega_{kj}(n) \qquad (44)
$$

- Combining these component partial derivatives we obtain

$$
\begin{aligned}
\frac{\partial E(n)}{\partial y_j(n)} &= -\sum_k \boxed{e_k(n)\varphi_k'(v_k(n))}\, \omega_{kj}(n) \\
&= -\sum_k \delta_k(n) \omega_{kj}(n) \qquad (45)
\end{aligned}
$$

Back-propagation Algorithm

- Substituting Equation (**??**) into Equation (**??**) to obtain

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n)\omega_{kj}(n) \tag{46}$$

and when combined with Equation (**??**) we can write the correction as

$$\begin{aligned}\Delta\omega_{ji}(n) &= \eta\delta_j(n)y_i(n) \\ &= \eta\varphi_j'(v_j(n)) \sum_k \delta_k(n)\omega_{kj}(n)y_i(n)\end{aligned} \tag{47}$$

and the update rule as

$$\begin{aligned}\omega_{ji}^{\text{new}}(n) &= \omega_{ji}^{\text{old}}(n) + \Delta\omega_{ji}(n) \\ &= \eta\delta_j(n)y_i(n) \\ &= \omega_{ji}^{\text{old}}(n) + \eta\varphi_j'(v_j(n)) \sum_k \delta_k(n)\omega_{kj}(n)y_i(n)\end{aligned} \tag{48}$$

which is the same expression we provided in Equation (**??**)

Summary of Back-propagation Algorithm for Multilayer Perceptron

1. Training could be Online (weight update after presentation of each sample) or Batch (weight update after presentation of all samples)
2. Back-propagation comprises two phases namely Forward pass and Backward pass
3. Forward pass: Weights of the network are fixed and input signal is propagated layer-wise through the network and transformed signal appears at the output; each neuron computes (see Figure (**??**))

$$v_j(n) = \sum_{j=0}^{m} \omega_{ji}(n) y_i(n); \quad y_j(n) = \varphi_j(v_j(n)) \tag{49}$$

4. In the Backward pass error is propagated backward through the network to compute weight updates (see Figure (**??**) and Equation (**??**)):

$$\omega_{ji}^{\text{new}}(n) = \omega_{ji}^{\text{old}}(n) + \begin{cases} \eta \boxed{e_j(n)\varphi_j'(v_j(n))}\, y_i(n) & \text{for output neurons} \\ \eta \boxed{\varphi_j'(v_j(n)) \sum_k \delta_k(n)\omega_{kj}(n)}\, y_i(n) & \text{for hidden neurons} \end{cases} \tag{50}$$

See handout on a simple hand calculation of back-propagation

Bibliography

Deep Learning. Ian Goodfellow and Yoshua Bengio and Aaron Courville. MIT press.2016

Neural Networks and Learning Machines, 3rd Edition. S.Haykin.2009