Multilayer Perceptron and Back-propagation Algorithm

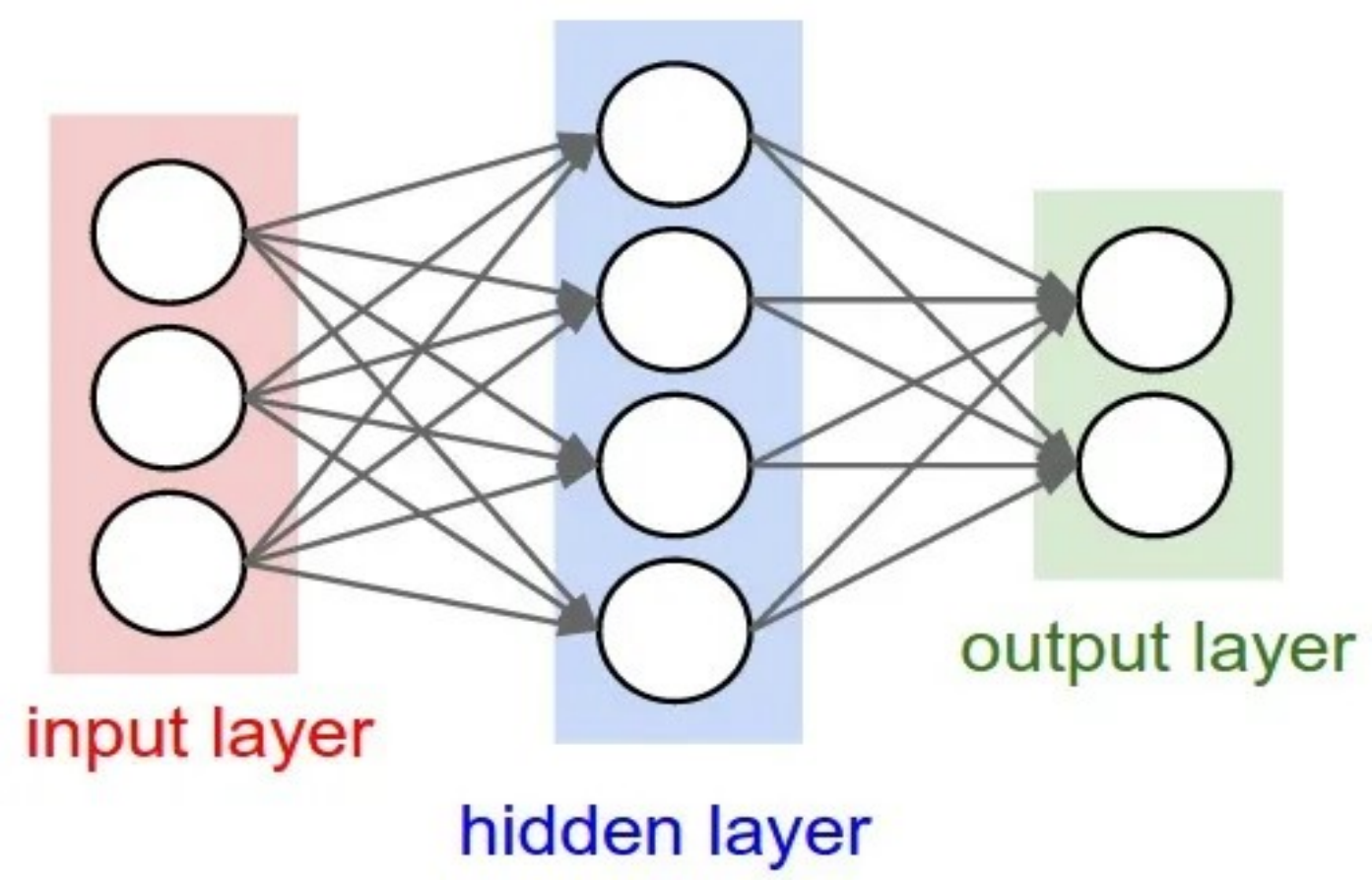
Multilayer Perceptron (MLP)

- A Multilayer Perceptron (MLP) is a type of feedforward neural network. It is widely used for regression and classification tasks in machine learning.
- Feedforward neural networks consist of multiple layers, where the layers are connected in a cascading fashion, with the output of one layer serving as the input to the next layer.
- Each layer consists of a certain number of neurons (or nodes), which is referred to as the layer's width. The number of layers in the network is referred to as its depth.

MLP (continued)

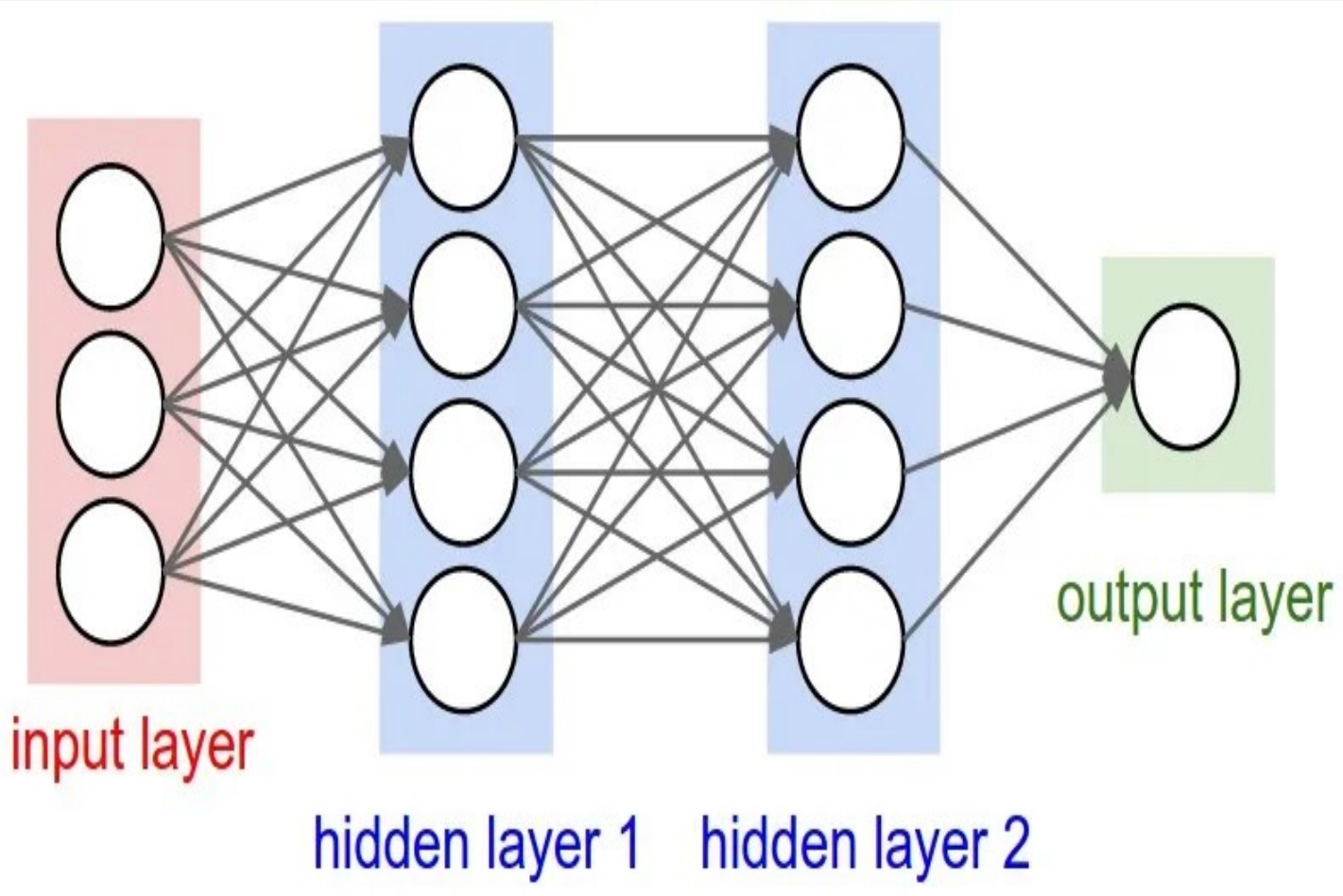
- In feedforward neural networks, every node in a layer is connected to all nodes in the next layer, propagating forward until the last layer. Hence, it is called a fully connected feedforward neural network.
- MLP consists of a set sensory nodes that constitute the input layer, one or more hidden layers of computation nodes, and an output layer of computation nodes.
- The input signal propagates through the network in a forward direction, on a layer-by-layer basis.

MLP (3:4:2) Network Structure



- Number of input neurons : 3
- Number of hidden neurons: 4
- Number of output neurons: 2

MLP (3:4:4:1) Network Structure



- Number of input neurons : 3
- Number of first hidden neurons : 4
- Number of first hidden neurons : 4
- Number of output neurons: 1

Network Architecture

- Layers: MLP consists of multiple layers of neurons.
- Input Layer: The initial layer that receives input data. Each neuron in this layer represents a feature of the input.
- Hidden Layers: One or more intermediate layers between the input and output layers. These layers perform computations and apply non-linear activation functions. Each neuron in these layers processes the inputs received from the previous layer.
- Output Layer: The final layer that produces the network's output. The number of neurons in this layer typically corresponds to the number of desired output values or classes.

Back-propagation Algorithm

- Except for the input nodes, each node is a neuron that uses a nonlinear activation function in other layers.
- MLP utilizes a supervised learning technique using back-propagation algorithm for training.
- Back-propagation learning consists of two passes through the different layers of the network: a forward pass and a backward pass.
- **Forward Pass:**
- Data flows through the network from the input layer through the hidden layers to the output layer. Each neuron in a layer applies its activation function to the weighted sum of its inputs, passing the result to the next layer.
- During the forward pass the weights and biases of the network are fixed.

Back-propagation learning (Continued)

- Backward pass:
- Error Calculation: The difference between the network's prediction and the actual target is computed using a loss function.
- Gradient Descent: Back-propagation calculates the gradient of the loss function with respect to each weight in the network. These gradients indicate how to adjust the weights to minimize the error.
- Weight Update: Weights are updated using optimization techniques such as gradient descent, stochastic gradient descent, or more advanced algorithms like Adam.
- The learning process performed to update weight parameters is known as **back-propagation learning**.

Training Phase

- **Epochs:** The network is trained over multiple iterations (epochs), where each epoch involves passing all training examples through the network, computing the error, and updating the weights.
- **Learning Rate:** A hyperparameter that controls the size of the weight updates during training. A proper learning rate is crucial for effective training.
- **Loss Functions:**
- Classification: Cross-entropy loss is commonly used to measure the performance of the network in classification task.
- Regression: Mean squared error (MSE) or mean absolute error (MAE) is used for regression tasks.

Suitable Loss Function for Classification

- In the context of Multi-Layer Perceptrons (MLPs) for classification, the cross-entropy loss function is generally preferred over the mean squared error (MSE) loss function. This is because of following reasons:
- **Cross-Entropy Loss:**
- Probability Output: Cross-entropy loss is well-suited for classification problems where the network outputs probabilities. It measures the difference between the predicted probability distribution and the true class labels.
- Logarithmic Penalty: It penalizes incorrect predictions more heavily, especially when the predicted probability for the true class is low. This is because it uses the logarithm of the predicted probability, which amplifies the penalty for incorrect predictions.

- Probability Interpretation: For classification, the network often uses a softmax activation function in the output layer to produce a probability distribution across classes. Cross-entropy loss is directly related to the probability distribution, making it a natural fit for such outputs.
- **Mean Squared Error Loss:**
- Regression Focused: MSE is more commonly used for regression tasks where the goal is to predict continuous values rather than probabilities.
- Less Sensitive to Probabilities: MSE does not handle probability outputs well because it treats the output values as continuous variables. It penalizes errors quadratically and is less sensitive to the probability distribution compared to cross-entropy.

- Lack of Logarithmic Penalty: MSE does not have the logarithmic component that can significantly impact the learning process, especially when dealing with probabilities close to zero or one.
- **Convergence and Gradient Behavior:**
- Better Gradient Behavior: Cross-entropy loss often provides more informative gradients during backpropagation, leading to faster and more stable convergence in classification problems.
- Effective Learning: It helps in faster convergence because it directly aligns with the objective of maximizing the likelihood of the correct class, which can lead to better optimization of the network weights.
- MSE can lead to slower convergence in classification tasks because it does not account for the probabilistic nature of the outputs. The gradients it produces can be less effective for updating weights in the classification context.

Notation used in Back-propagation

- The indices i, j, and k refer to different neurons in the network; with signals propagating through the network from left to right, neuron j lies in a layer to the right of neuron i, and neuron k lies in a layer to the right of neuron j when neuron j is a hidden unit.
- In iteration (time step) n, the nth training pattern (example) is presented to the network.
- The symbol $\mathcal{E}(n)$ refers to the instantaneous sum of error squares or error energy at iteration n. The average of $\mathcal{E}(n)$ over all values of n (i.e., the entire training set) yields the average error energy $\mathcal{E}_{av}$.
- The symbol $e_j(n)$ refers to the error signal at the output of neuron j for iteration n.
- The symbol $d_j(n)$ refers to the desired response for neuron j and is used to compute $e_j(n)$.
- The symbol $y_j(n)$ refers to the function signal appearing at the output of neuron j at iteration n.
- The symbol $w_{ji}(n)$ denotes the synaptic weight connecting the output of neuron i to the input of neuron j at iteration n. The correction applied to this weight at iteration n is denoted by $\Delta w_{ji}(n)$.

- The induced local field (i.e., weighted sum of all synaptic inputs plus bias) of neuron j at iteration n is denoted by $v_j(n)$; it constitutes the signal applied to the activation function associated with neuron j.
- The activation function describing the input–output functional relationship of the nonlinearity associated with neuron j is denoted by $\varphi_j(\cdot)$.
- The bias applied to neuron j is denoted by b_j; its effect is represented by a synapse of weight $w_{j0} = b_j$ connected to a fixed input equal to +1.
- The ith element of the input vector (pattern) is denoted by $x_i(n)$.
- The kth element of the overall output vector (pattern) is denoted by $o_k(n)$.
- The learning-rate parameter is denoted by η.
- The symbol m_l denotes the size (i.e., number of nodes) in layer l of the multilayer perceptron; $l = 0, 1, \ldots, L$, where L is the "depth" of the network. Thus m_0 denotes the size of the input layer, m_1 denotes the size of the first hidden layer, and m_L denotes the size of the output layer. The notation $m_L = M$ is also used.

Back-propagation Algorithm

The error signal at the output of neuron j at iteration n (i.e., presentation of the nth training example) is defined by

$$e_j(n) = d_j(n) - y_j(n), \qquad \text{neuron } j \text{ is an output node} \tag{4.1}$$

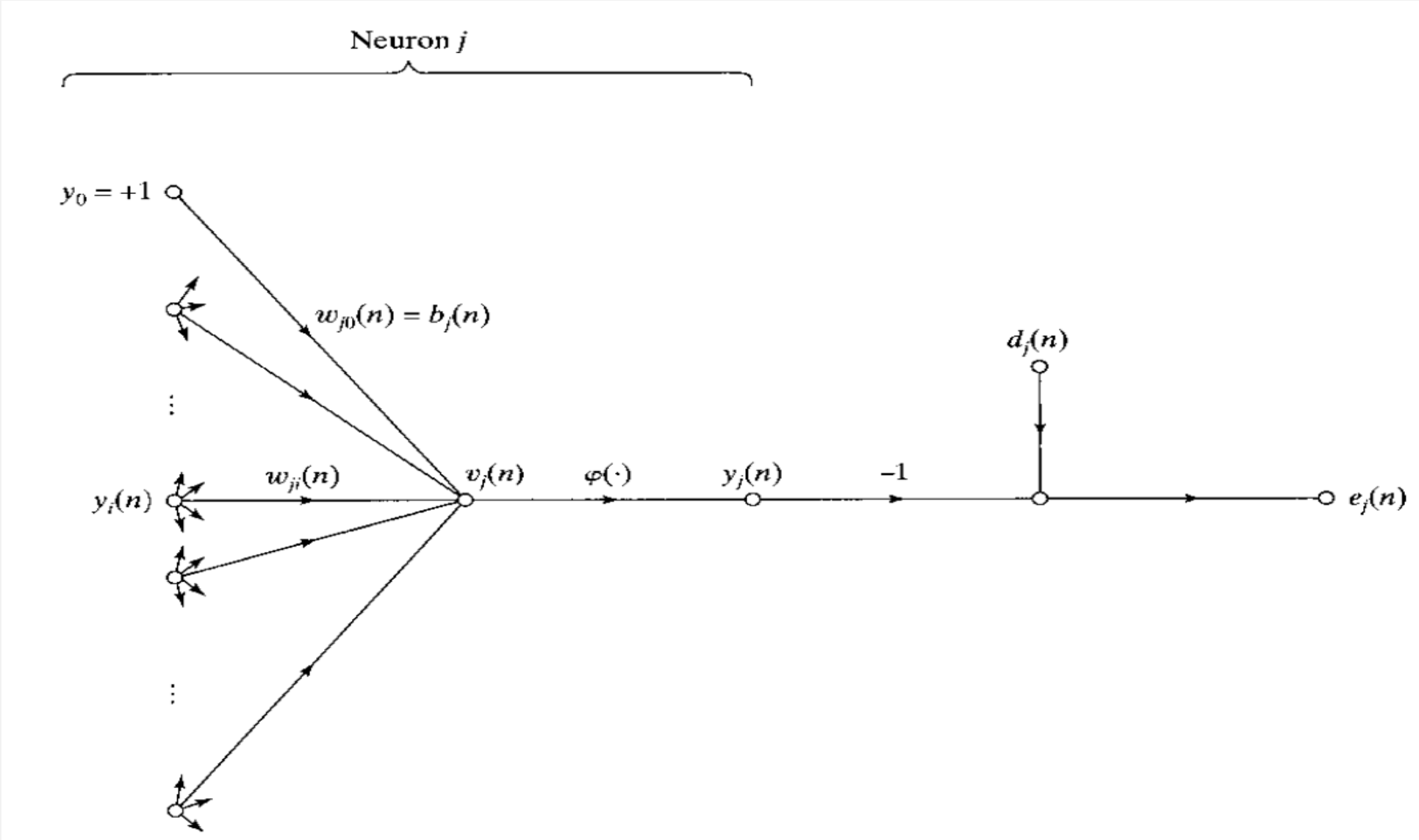
We define the instantaneous value of the error energy for neuron j as $\frac{1}{2}e_j^2(n)$. Correspondingly, the instantaneous value $\mathscr{E}(n)$ of the total error energy is obtained by summing $\frac{1}{2}e_j^2(n)$ over *all neurons in the output layer;* these are the only "visible" neurons for which error signals can be calculated directly. We may thus write

$$\mathscr{E}(n) = \frac{1}{2}\sum_{j \in C} e_j^2(n) \tag{4.2}$$

where the set C includes all the neurons in the output layer of the network. Let N denote the total number of patterns (examples) contained in the training set. The *average squared error energy* is obtained by summing $\mathscr{E}(n)$ over all n and then normalizing with respect to the set size N, as shown by

$$\mathscr{E}_{\text{av}} = \frac{1}{N}\sum_{n=1}^{N} \mathscr{E}(n) \tag{4.3}$$

- The objective of the learning process is to adjust the free parameters of the network to minimize the loss function.



In a manner similar to the LMS algorithm, the back-propagation algorithm applies a correction $\Delta w_{ji}(n)$ to the synaptic weight $w_{ji}(n)$, which is proportional to the partial derivative $\partial \mathscr{E}(n)/\partial w_{ji}(n)$. According to the *chain rule* of calculus, we may express this gradient as:

$$\frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} = \frac{\partial \mathscr{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)} \tag{4.6}$$

The partial derivative $\partial \mathscr{E}(n)/\partial w_{ji}(n)$ represents a *sensitivity factor*, determining the direction of search in weight space for the synaptic weight w_{ji}.

Differentiating both sides of Eq. (4.2) with respect to $e_j(n)$, we get

$$\frac{\partial \mathscr{E}(n)}{\partial e_j(n)} = e_j(n) \tag{4.7}$$

Differentiating both sides of Eq. (4.1) with respect to $y_j(n)$, we get

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \tag{4.8}$$

Next, differentiating Eq. (4.5) with respect to $v_j(n)$, we get

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n)) \tag{4.9}$$

where the use of prime (on the right-hand side) signifies differentiation with respect to the argument. Finally, differentiating Eq. (4.4) with respect to $w_{ji}(n)$ yields

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n) \tag{4.10}$$

The use of Eqs. (4.7) to (4.10) in (4.6) yields

$$\frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n) \tag{4.11}$$

The correction $\Delta w_{ji}(n)$ applied to $w_{ji}(n)$ is defined by the *delta rule*:

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} \tag{4.12}$$

where η is the *learning-rate parameter* of the back-propagation algorithm. The use of the minus sign in Eq. (4.12) accounts for *gradient descent* in weight space (i.e., seeking a direction for weight change that reduces the value of $\mathscr{E}(n)$). Accordingly, the use of Eq. (4.11) in (4.12) yields

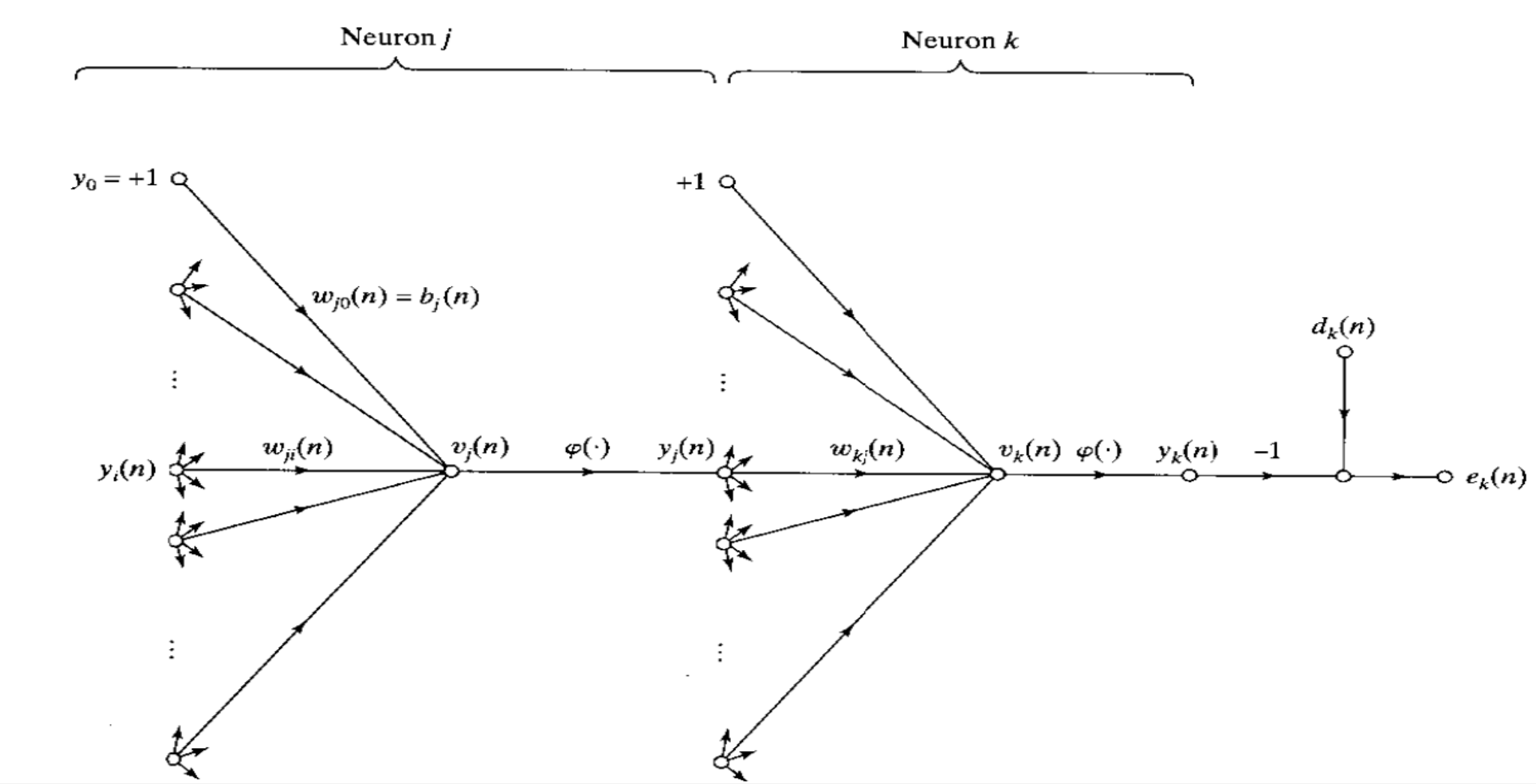
$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n) \tag{4.13}$$

where the *local gradient* $\delta_j(n)$ is defined by

$$\begin{aligned} \delta_j(n) &= -\frac{\partial \mathscr{E}(n)}{\partial v_j(n)} \\ &= -\frac{\partial \mathscr{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \\ &= e_j(n) \varphi_j'(v_j(n)) \end{aligned} \tag{4.14}$$

The local gradient points to required changes in synaptic weights. According to Eq. (4.14), the local gradient $\delta_j(n)$ for output neuron j is equal to the product of the corresponding error signal $e_j(n)$ for that neuron and the derivative $\varphi_j'(v_j(n))$ of the associated activation function.

Weights Correction of Hidden Layer neurons ($i -> j -> k$)



Chain Rule of Calculus for Hidden Layer Neurons

- $w_{ji} \rightarrow$
  $weight\ parameter\ for\ jth\ neuron\ of\ hidden\ layer\ to\ ith\ neuron\ of\ input\ layer$

$$\frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} = \frac{\partial \mathscr{E}(n)}{\partial y_k(n)} \frac{\partial y_k(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial w_{ji}(n)}$$

- $\frac{\partial \mathscr{E}(n)}{\partial y_k(n)} = \frac{\partial \mathscr{E}(n)}{\partial e_k(n)} \frac{\partial e_k(n)}{\partial y_k(n)}$
- $\frac{\partial y_k(n)}{\partial y_j(n)} = \frac{\partial y_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)}$
- $\frac{\partial y_j(n)}{\partial w_{ji}(n)} = \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)}$

$$\frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} = \frac{\partial \mathscr{E}(n)}{\partial e_k(n)} \frac{\partial e_k(n)}{\partial y_k(n)} \frac{\partial y_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

Weight (w_{ji}) Correction of Hidden Layer

- w_{ji} is the weight for the j^{th} neuron of hidden layer to i^{th} neuron of input layer connection.
- $\frac{\partial \mathcal{E}(n)}{\partial e_k(n)} = \sum_k e_k \quad \frac{\partial e_k(n)}{\partial y_k(n)} = -1 \quad \frac{\partial y_k(n)}{\partial v_k(n)} = \varphi\left(v_k'(n)\right) \quad \frac{\delta v_k(n)}{\delta y_j(n)} = w_{kj}(n)$
- $\frac{\partial \mathcal{E}(n)}{\delta y_j(n)} = -\sum_k e_k \varphi\left(v_k'(n)\right) w_{kj}(n) = -\sum_k \delta_k(n) w_{kj}(n)$
- $\delta_k(n) = e_k \varphi\left(v_k'(n)\right)$ **// Local gradient for kth output neuron**
- $\delta_j(n) = \varphi_j\left(v_j'(n)\right) \sum_k \delta_k(n) w_{kj}(n)$**// Local gradient for jth hidden neuron**

Derivative of Activation Functions

- The computation of the δ (local gradient) for a each neuron of the multilayer perceptron requires derivative of the activation function $\varphi'(.)$ associated with that neuron.
- A continuously differentiable nonlinear activation function commonly used in multilayer perceptron.
- Sigmoid or Logistic Activation Function:

$$\varphi_j\big(v_j(n)\big) = \frac{1}{1+e^{-v_j(n)}}$$

$$\varphi_j'\big(v_j(n)\big) = \frac{e^{-v_j(n)}}{1+e^{-v_j(n)}}$$

As we know $y_j(n) = \varphi_j\big(v_j(n)\big)$ then we simplified above equation as

$$\boxed{\varphi_j'\big(v_j(n)\big) = y_j(n)(1 - y_j(n))}$$

The correction $\Delta w_{ji}(n)$ applied to $w_{ji}(n)$ is defined by the *delta rule*:

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)}$$

Weight correction $\Delta w_{ji}(n)$ using local gradient $\delta_j(n)$

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n)$$

$$\begin{pmatrix} \text{Weight} \\ \text{correction} \\ \Delta w_{ji}(n) \end{pmatrix} = \begin{pmatrix} \text{learning-} \\ \text{rate parameter} \\ \eta \end{pmatrix} \cdot \begin{pmatrix} \text{local} \\ \text{gradient} \\ \delta_j(n) \end{pmatrix} \cdot \begin{pmatrix} \text{input signal} \\ \text{of neuron } j \\ y_i(n) \end{pmatrix}$$

Note: As learning rate (η) controls the convergence speed of the back-propagation learning algorithm like smaller (η), the smaller changes to the synaptic weights in the training that leads to smother trajectory in weight space. On the other hand, if we make (η) too large in order to speed of the rate of learning, as a result large change in weights and network becomes unstable (i.e., oscillatory). A simple method to increase the rate of learning yet to avoid the danger of instability is to modify delta rule by including a momentum term named as generalized delta rule, where α is usually a positive number called the momentum constant.

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n)$$

Back-propagation Learning Algorithm

1. Initialization: Randomly initialize weights and biases in the range of (-1 to 1).Normalize the dataset within the range of (0 to 1) or (-1 to 1). Initialize learning rate (η) and momentum (α) hyperparameters with values in the range (0 to 1). Finally, set number of iterations or epochs to be trained depends on dataset.

2. Presentation of Training Examples: Divide the dataset into two sets {training set and test set} and present randomly all normalized training samples in each epoch for forward and backward computations in step-3 and step-4 respectively.

3. Forward Computation:

3. *Forward Computation.* Let a training example in the epoch be denoted by $(\mathbf{x}(n), \mathbf{d}(n))$, with the input vector $\mathbf{x}(n)$ applied to the input layer of sensory nodes and the desired response vector $\mathbf{d}(n)$ presented to the output layer of computation nodes. Compute the induced local fields and function signals of the network by proceeding forward through the network, layer by layer. The induced local field $v_j^{(l)}(n)$ for neuron j in layer l is

$$v_j^{(l)}(n) = \sum_{i=0}^{m_0} w_{ji}^{(l)}(n) y_i^{(l-1)}(n)$$

where $y_i^{(l-1)}(n)$ is the output (function) signal of neuron i in the previous layer $l-1$ at iteration n and $w_{ji}^{(l)}(n)$ is the synaptic weight of neuron j in layer l that is fed from neuron i in layer $l-1$. For $i = 0$, we have $y_0^{(l-1)}(n) = +1$ and $w_{j0}^{(l)}(n) = b_j^{(l)}(n)$ is the bias applied to neuron j in layer l. Assuming the use of a sigmoid function, the output signal of neuron j in layer l is

$$y_j^{(l)} = \varphi_j(v_j(n))$$

If neuron j is in the first hidden layer (i.e., $l = 1$), set

$$y_j^{(0)}(n) = x_j(n)$$

where $x_j(n)$ is the jth element of the input vector $\mathbf{x}(n)$. If neuron j is in the output layer (i.e., $l = L$, where L is referred to as the *depth* of the network), set

$$y_j^{(L)} = o_j(n)$$

Compute the error signal

$$e_j(n) = d_j(n) - o_j(n)$$

where $d_j(n)$ is the jth element of the desired response vector $\mathbf{d}(n)$.

4. Backward Computation:

4. *Backward Computation.* Compute the δs (i.e., local gradients) of the network, defined by

$$\delta_j^{(l)}(n) = \begin{cases} e_j^{(L)}(n)\,\varphi_j'(v_j^{(L)}(n)) & \text{for neuron } j \text{ in output layer } L \\ \varphi_j'(v_j^{(l)}(n)) \sum_k \delta_k^{(l+1)}(n) w_{kj}^{(l+1)}(n) & \text{for neuron } j \text{ in hidden layer } l \end{cases}$$

where the prime in $\varphi_j'(\cdot)$ denotes differentiation with respect to the argument. Adjust the synaptic weights of the network in layer l according to the generalized delta rule:

$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \alpha[w_{ji}^{(l)}(n-1)] + \eta\delta_j^{(l)}(n) y_i^{(l-1)}(n)$$

where η is the learning-rate parameter and α is the momentum constant.

5. Iteration: Iterate the forward and backward computations under step-3 and step-4 by presenting new epochs of training examples to the network until the stopping criterion is met.

Note: The order of presentation of training examples should be randomized from epoch to epoch.